华章科技

系统归纳和深刻解读PHP开发中的编程思想、底层原理、核心技术、开发技巧、编码规范和最佳实践，为PHP程序员进阶修炼提供全面而高效的指导！

實戰

繁体版台湾发行

列旭松 陈文 著

PHP Core Technology and Best Practice

PHP
核心技术与最佳实践

机械工业出版社
China Machine Press

华章科技

系统归纳和深刻解读PHP开发中的编程思想、底层原理、核心技术、开发技巧、编码规范和最佳实践，为PHP程序员进阶修炼提供全面而高效的指导！

實戰

繁体版台湾发行

列旭松 陈文 著

PHP Core Technology and Best Practice

PHP
核心技术与最佳实践

机械工业出版社
China Machine Press

前言

为什么要写这本书

近几年，市场上关于PHP的书已经很多了，各种培训机构也如雨后春笋般不断增加。那为什么还要写这本书呢？这本书存在的意义又在哪里？这要从下面的几个问题说起。

有没有这样一本PHP教材，它不讲HTML和CSS，也不讲JavaScript基础，甚至不讲PHP语法基础？

有没有这样一本PHP教材，它不讲留言本或博客的开发，也不讲数据库的CRUD操作？

有没有这样一本PHP教材，它专注于Web开发技术的最前沿，深入浅出，适合中高级程序员的进阶和提高？

有没有这样一本PHP教材，它提倡面向对象的程序思想，提倡算法和数据结构的重要性，提倡对网络协议的深入理解，且没有大篇幅的代码，而是更多偏重于理论讲解？

有没有这样一本PHP教材，它探讨PHP的扩展开发，探讨高并发大流量的架构，深入探讨NoSQL的内部实现和细节？

以上几个问题也是我在早期PHP学习的过程中一直在寻找的答案，可是我并没有找到一本理想的PHP书籍，一本适合中高级程序员进阶的书籍。当怀着同样问题的旭松兄找到我时，我们不禁产生一个念头：“既然现在市场上缺少一本这样的书籍，我们何不自己写一本呢？利己利人的事值得去做。”然后一拍即合，说做就做，现在这本书经历长达一年多的酝酿和写作过程终于完稿了。

我是在大学期间接触到PHP语言的，并马上被其简洁的语法和极高的开发效率所吸引，一头扎进PHP开发的世界中。随着学习的深入，并经常关注PHP社区的动态，我很快意识到一些PHP社区普遍存在的问题。比如PHP社区一直争论算法重不重要，面向对象好不好，代码质量重要还是开发速度重要的问题。还有譬如为什么我去大型互联网公司应聘PHP程序员，却不考察我对PHP语法和函数的掌握情况，而是会问我

C语言、算法、网络协议、高并发处理、MVC理论这些看似和PHP不沾边的问题。

PHP到底要怎么学，学什么，一个高级PHP程序员应该是什么样的，我想这也是很多PHP新手和工作一两年的PHP开发者的疑惑。这本书所要解决的就是这一系列的问题。

在我看来，一本技术书籍的价值在于其对知识的提炼和与众不同的地方。举例来说，到一个书店去看书，你最想用笔抄下来或撕下来带走的那几页，就是对你帮助最大的东西，也是你认为这本书的价值所在。也是基于这个想法，我们思考这本书该写什么，怎么写，哪些地方对读者有帮助。我们试图从不同的角度带领读者来看PHP，进而给这本书注入一些不一样的东西。我们希望这是一件有意义的事。

本书适合的对象

PHP爱好者；

想进阶的初级PHP程序员；

对PHP扩展开发感兴趣的读者；

对高并发感兴趣的读者；

对NoSQL应用和实现原理感兴趣的读者；

从事PHP网络应用，想知道HTTP协议、Socket等更多细节的开发人员；

想就职于大型互联网公司的PHP程序员；

开设相关课程的大专院校的学生；

公司内部培训的学员。

如何阅读本书

本书一共有14章。每章节都可以单独阅读，由于部分知识点之间存在一定的衔接，故建议按先后顺序阅读。

第1章　为面向对象思想的核心概念。本章主要讲解面向对象开发的思想，重点讲解面向对象模型的建立，以及面向对象的一些基础概念。通过大量对比和实例，尤其是与Java的对比，力图从不同角度讲解PHP面向对象的特性，让PHP程序员看到不同的面向对象。求同存异是本章的核心思想。

第2章　为用面向对象思维写程序。本章用简练的语言讲解了面向对象设计的五大原则，这五大原则也是理解设计模式的基础所在，帮助读者站在一个更高的角度思考面向对象。

第3章　为正则表达式技巧与实战。本章详细介绍了正则的基础语法，通过大量的示例、通俗的语言讲解正则概念，引导读者理解正则的一系列规则。接下来，结合实际工作用安全过滤、URL重写等实例，加深对正则的应用和掌握。最后给出正则效率优化的一些普遍技巧和替代方案，让读者对正则的使用得心应手。

第4章　为PHP网络技术及应用。本章着重介绍了HTTP协议、Socket开发、WebService、Cookie和Session使用等。结合实战向读者阐述网络开发的核心和重点，特别是对HTTP协议的理解。HTTP协议是Web开发的基石，也是各种面试和开发中必然遇到的知识点。而Socket则是应用交互的桥梁，保证了有用的可扩展性。

第5章　为PHP与数据库基础。本章从不同角度分析了MySQL，介绍了PDO、MySQL优化、存储过程、事件调度机制以及MySQL安全防范等内容。

第6章　为PHP模板引擎的原理与最佳实践。本章通过实现一个简单的模板引擎，学习模板引擎的原理和使用方法，然后对比几大流行的模板引擎实现方案，简单介绍了各种实现方案的思想和优缺点，最后探讨模板引擎的意义。

第7章　为PHP扩展开发。本章的知识是本书核心内容，介绍了PHP扩展开发的几个重要知识点，如扩展框架搭建、PHP生命周期、PHP变量在内核中的实现方式、Zend引擎、内存管理等，让读者深入PHP底层，知其然也知其所以然。

第8章　为缓存。本章主要介绍了缓存的基本原理和三个衡量指标，通过几个实例加深读者对缓存的理解。利用本章知识，读者应该能

设计一个比较合理的缓存方案。

第9章　为Memcached应用与内幕。本章深入剖析了Memcached的实现和内部结构，从而使读者掌握Memcached的高级应用，对构建复杂环境的缓存层有个清晰的认识。

第10章　为Redis应用与内幕。本章重点介绍了Redis的深入应用，如事务处理、主从同步、虚拟内存等，和第9章类似，探讨了Redis的实现内幕。合理利用Redis可以为我们解决大流量高并发的应用。

第11章　为高性能网站架构。本章探讨了高性能架构的基本出发点，重点以HandlerSocket、MySQL主从复制、反向代理缓存软件Varnish和任务分发框架Gearman为例，讲述几种高性能架构中会用到的技术。

第12章　为调试与测试。科学的调试方法有助于快速找出潜在的Bug、理解复杂应用的流程、提高开发效率。单元测试是代码质量的保障。在这一章的最后一节介绍了使用JMeter进行压力测试的方法。

第13章　为Hash算法与数据库的实现。本章介绍了Hash算法的基本原理，用此算法实现一个简单的、基于Hash的数据库，让读者意识到算法的重要性和可操作性。

第14章　为PHP编码规范。本章介绍了PHP开发中应遵循的基本代码规范，并提出合理建议。好的代码必然是规范的代码。

本书第1、2、3、5、6、8、12、14章由陈文撰写，第7、9、10、11、13章由列旭松撰写，第4章由两人共同完成。

勘误和支持

由于我们的水平和开发经验有限，同时计算机技术更新较快，书中难免存在不足之处，有些章节内容可能从未来的某一天开始不再适用，还望读者理解和体谅，并恳请读者批评指正。您若对本书有什么好的建议或者对书中部分内容有疑惑，可与我们联系，我们将尽量为读者提供最满意的解答。期待得到您的真挚反馈。我们的联系方式如下：

陈文：waitfox@qq.com

列旭松：liexusong@qq.com

感谢

首先要感谢PHP之父Rasmus Lerdorf，是他创建了这个简单、轻松、有趣、快速而又高效的语言；其次，感谢PHP社区每一位充满活力的朋友，和你们的交流使我学到很多，本书有不少内容就来自于社区的智慧。

在这里尤其要感谢机械工业出版社华章公司的大力支持，特别是杨福川和白宇两位编辑，在一年多的时间里，因为有了你们的耐心指导、逐字逐句认真审稿和改稿才有了本书的诞生。

最后，还要感谢家人和朋友的支持。

陈文

第1章　面向对象思想的核心概念

面向对象是什么？以下是维基百科对面向对象的解释：

面向对象程序设计（Object Oriented Programming, OOP）是一种程序设计范型，同时也是一种程序开发方法。它将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和可扩展性。

面向过程、面向对象以及函数式编程被人们称为编程语言中的三大范式（实际上，面向过程与面向对象都同属于命令式编程），是三种不同编码和设计风格。其中面向对象的核心思想是对象、封装、可重用性和可扩展性。

面向对象是一种更高级、更抽象的思维方式，面向过程虽然也是一种抽象，但面向过程是一种基础的抽象，面向对象又是建立在面向过程之上的更高层次的抽象，因此对面向对象的理解也就不是那么容易了。

面向对象和具体的语言无关。在面向对象的世界里，常常提到的两种典型语言——C++和Java。它们都是很好的面向对象的开发语言。实际上，像C语言这种大家普遍认为的面向过程开发的主打语言，也能进行面向对象的开发，就连JavaScript这门很久之前一直被视作面向过程编程的语言，人们对它的认识也发生了改变，逐渐承认其是面向对象的语言，并且也接受了JavaScript独特的面向对象的语法。所以我们说面向对象只是种程序设计的理念，和具体的语言无关。不同的程序员既可以用C语言写出面向对象的风格来，也可以用Java写成面向对象的风格。这里并不是说面向对象的风格要优于面向过程，而是二者各有自己所擅长的领域。OOPL（Object Oriented Programming Language）可以提高程序的封装性、复用性、可维护性，但仅仅是“可以”，能不能真正实现这些优点，还取决于编程和设计人员。就PHP而言，其不是一门纯的面向对象的语言，但是仍然可以使用PHP写出好的面向对象风格的代码。

实际开发中，面向对象为什么让我们觉得那么难？面向对象究竟难在什么地方？为什么面向对象开发在PHP里一直不是很受重视，并且没有得到普及和推广？PHP对面向对象的支持到底如何？怎么学习面向对象的思维？

在这里，我们将就面向对象一些概念展开讨论，其中重点讨论PHP特色的面向对象的风格和语法，并通过相互借鉴和对比，使读者认识PHP自身的特点，尤其是和其他语言中不同的地方。

1.1 面向对象的“形”与“本”

类是对象的抽象组织，对象是类的具体存在。

2200年前的战国时期，赵国平原君的食客公孙龙在骑着白马进城时，被守城官以马不能入城拦下，公孙龙即兴演讲，口述“白马非马”一论，守城官无法反驳，于是公孙龙就骑着他的白马（不是马的）进城去了。这就是历史上最经典的一次对面向对象思维的阐述。

公孙龙的“白马非马”论如下：

“白马非马”，可乎？曰：“可。”曰“何哉？”曰：“马者，所以命形也；白者，所以命色也。命色者非命形也。故曰：‘白马非马’。”曰：“有白马不可谓无马也。不可谓无马者，非马也？有白马为有马，白之，非马何也？”曰：“求马，黄、黑马皆可致；求白马，黄、黑马不可致。使白马乃马也，是所求一也。所求一者，白者不异马也。所求不异，如黄、黑马有可有不可，何也？可与不可，其相非明。故黄、黑马一也，而可以应有马，而不可以应有白马，是白马之非马，审矣！”

公孙龙乃战国时期的“名家”，名家的中心论题是所谓“名”（概念）和“实”（存在）的逻辑关系问题。名者，抽象也，类也。实者，具体也，对象也。从这个角度讲，公孙龙是我国早期的最著名的面向对象思维的思想家。

“白马非马”这一论断的关键就在于“非”字，公孙龙一再强调白马与马的特征，通过把白马和马视为两个不同的类，用“非”这一关系，成功地把“白马”与“马”的关系由从属关系转移到“白马”这个对象与“马”这个对象的相等关系上，显然，二者不等，故“白马非马”。而我们正常的思维是，马是一个类，白马是马这个类的一个对象，二者属于从属关系。说“白马非马”，就是割裂马与白马之间的从属关系，偷换概念，故为诡辩也。

白马非马这个典故，我们可以称之为诡辩。但我们把这个问题抽象

出来，实际上讨论的就是类与类之间的界定、类的定义等一系列问题，类应该抽象到什么程度，其中即涉及了类与对象的本质问题，也涉及了类的设计过程中的一些原则。

1.1.1 对象的“形”

要回答类与对象本质这个问题，我想可以先从“形”的角度来回答。本节以PHP为例，来探讨对象的“形”与“本”的问题。

类是我们对一组对象的描述。

在PHP里，每个类的定义都以关键字class开头，后面跟着类名，紧接着一对花括号，里面包含有类成员和方法的定义。如下面代码所示：

```
class person {
public $name;
public $gender;
public function say() {
echo $this->name, "is", $this->gender;
}
}
```

在这里，我们定义了一个person类。代表了抽象出来的人这个概念，它含有姓名和性别这两个属性，还具有一个开口说话的方法，这个方法会告诉外界这个人的性别和姓名。我们接下来就可以产生这个类的实例：

```
$student=new person();
$student->name='Tom';
$student->gender='male';
$student->say();
$teacher=new person();
$teacher->name='Kate';
$teacher->gender='female';
$teacher->say();
```

这段代码则实例化了person类，产生了一个student对象和teacher对象的实例。实际上也就是从抽象到具体的过程。现实世界中，仅仅说“人”是没有意义的，中国人把它叫“人”，美国人把它叫person或者human，如果高兴，把它叫“God”或者“板凳”都无所谓。但是只要你把“人”这个概念加上各种属性和方法，比如说有两条腿、直立行走、会说话，则无论是中国人，还是美国人，甚至外星人都是能理解你所描述的事物。所以，一个类的设计需要能充分展示其最重要的属性和方法，并且能与其他事物相区分。只有类本身有意义，从抽象到具体的实例化

才会有意义。

根据上面的实例代码，可以有下面的一些理解：

类定义了一系列的属性和方法，并提供了实际的操作细节，这些方法可以用来对属性进行加工。

对象含有类属性的具体值，这就是类的实例化。正是由于属性的不同，才能区分不同的对象。在上面例子里，由于student和teacher的性别和姓名不一样，才得以区分开二人。

类与对象的关系类似一种服务与被服务、加工与被加工的关系，具体而言，就如同原材料与流水线的关系。只需要在对象上调用类中所存在的方法，就可以对类的属性进行加工，并且展示其功能。

类是属性和方法的集合，那么在PHP里，对象是什么呢？比较普遍的说法就是“对象由属性和方法组成”。对象是由属性组成，这很好理解，一个对象的属性是它区别于另一个对象的关键所在。由于PHP的对象是用数组来模拟的，因此我们把对象转为数组，就能看到这个对象所拥有的属性了。

继续使用上面代码，可以打印student对象：

```
print_r((array)$student);
var_dump($student);
```

到这里，可以很直观地认识到，对象就是一堆数据。既然如此，可以把一个对象存储起来，以便需要时用。这就是对象的序列化。

所谓序列化，就是把保存在内存中的各种对象状态（属性）保存起来，并且在需要时可以还原出来。下面的代码实现了把内存中的对象当前状态保存到一个文件中：

```
$str=serialize($student);
echo$str;
file_put_contents('store.txt',$str);
```

输出序列化后的结果：

```
O:6:"person":2:{s:4:"name";s:3:"Tom";s:6:"gender";s:4:"mail";}
```

在需要时，反序列化取出这个对象：

```
$str=file_get_contents('store.txt');
$student=unserialize($str);
$student->say();
```

注意　在序列化和反序列化时都需要包含类的对象的定义，不然有可能出现在反序列化对象时，找不到该对象的类的定义，而返回不正确的结果。

可以看到，对象序列化后，存储的只是对象的属性。类是由属性和方法组成的，而对象则是属性的集合，由同一个类生成的不同对象，拥有各自不同的属性，但共享了类的代码空间中方法区域的代码。

1.1.2 对象的“本”

我们需要更深入地了解这种机制，看对象的“本”。对象是什么？对象在PHP中也是变量的一种，所以先看PHP源码中对变量的定义：

```
#zend/zend.h
typedef union _zvalue_value {
long lval; /*long value*/
double dval; /*double value*/
struct {
char *val;
int len;
} str;
HashTable *ht; /*hash table value*/
zend_object_value obj;
} zvalue_value;
```

zvalue_value，就是PHP底层的变量类型，zend_object_value obj就是变量中的一个结构。接着看对象的底层实现。

在PHP5中，对象在底层的实现是采取“属性数组+方法数组”来实现的。可以简单地理解为PHP对象在底层的存储如图1-1所示。

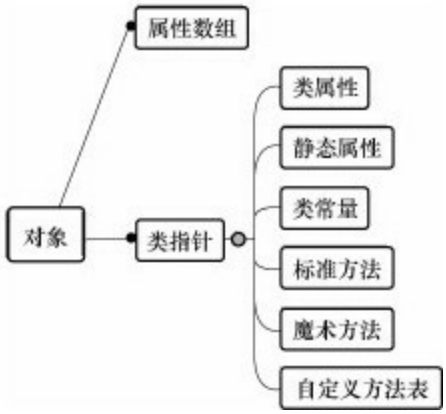

图 1-1 对象的组成

对象在PHP中是使用一种zend_object_value结构体来存储的。对象在ZEND（PHP底层引擎，类似Java的JVM）中的定义如下：

```
#zend/zend.h
typedef struct _zend_object {
zend_class_entry *ce; //这里就是类入口
HashTable *properties; //属性组成的HashTable
HashTable *guards; /*protects from get/set……recursion*/
} zend_object;
```

ce是存储该对象的类结构，在对象初始化时保存了类的入口，相当

于类指针的作用。prop　　erties是一个HashTable，用来存放对象属性。guards用来阻止递归调用。

类的标准方法在zend/zend__object__handlers.h文件中定义，具体实现则是在zend/zend__

object__handlers.　c文件中。关于PHP变量的存储结构的底层实现，将在第7章中进行更深入的介绍。

通过对上述源代码的简单阅读，可以更清晰地认识到对象也是一种很普通的变量，不同的是其携带了对象的属性和类的入口。

1.1.3 对象与数组

对象是什么，我们不好理解，也不容易回答，但是我们知道数组是什么。数组的概念比较简单。可以拿数组和对象对比来帮助我们理解对象。对象转化为数组，数组也能转换成对象。数组是由键值对数据组成的，数组的键值对和对象的属性/属性值对十分相似。对象序列化后和数组序列化后的结果是惊人的相似。如下面的代码所示：

```
$student_arr=array('name'=>'Tom','gender'=>'male');
echo"\n";
echo serialize($student_arr);
```

输出为：

```
a:2:{s:4:"name";s:3:"Tom";s:6:"gender";s:4:"male";}
```

可以很清楚地看出，对象和数组在内容上一模一样！

而对象和数组的区别在于：对象还有个指针，指向了它所属的类。在对student对象序列化时，我们看到了“person”这几个字符，这个标识符就标志了这个对象归属于person类，故在取出这个对象后，可以立即对其执行所包含的方法。如果对象中还包含对象呢？我们来看下一节的内容。

1.1.4 对象与类

在前面代码中定义了一个类，并创建了这个类的对象，把前面产生的对象作为这个新对象的一个属性，完整代码如代码清单1-1所示。

代码清单1-1 object.php

```
<? php
class person {
public $name;
public $gender;
public function say () {
echo $this->name, "\tis", $this->gender, "\r\n";
}
}
class family {
public $people;
public $location;
public function construct ( $p, $loc ) {
$this->people=$p;
$this->location=$loc;
}
}
$student=new person ();
$student->name=' Tom' ;
$student->gender=' male' ;
$student->say ();
$tom=new family ( $student, ' peking' );
echo serialize ( $student );
$student__arr=array (' name' =>' Tom' , ' gender' =>' male' );
echo"\n";
echo serialize ( $student__arr );
print__r ( $tom );
echo"\n";
echo serialize ( $tom );
```

输出结果如下：

```
Tom  is male
O: 6: "person": 2: {s: 4: "name"; s: 3: "Tom"; s: 6: "gender"; s: 4: "male"; }
a: 2: {s: 4: "name"; s: 3: "Tom"; s: 6: "gender"; s: 4: "male"; }
family Object
(
[people]=>person Object
(
[name]=>Tom
[gender]=>male
)
[location]=>peking
)
O: 6: "family": 2: {s: 6: "people"; O: 6: "person": 2: {s: 4: "name"; s: 3: "Tom"; s: 6: "gender"; s: 4: "male"; } s: 8: "
location"; s: 6: "peking"; }
```

可以看出，序列化后的对象会附带所属的类名，这个类名保证此对象能够在执行类的方法（也是自己所能执行的方法）时，可以正确地找到方法所在的代码空间（即对象所拥有的方法存储在类里）。另外，当一个对象的实例变量引用其他对象时，序列化该对象时也会对引用对象进行序列化。

基于如上的分析，可以总结出对象和类的概念以及二者之间的关

系：

类是定义一系列属性和操作的模板，而对象则把属性进行具体化，然后交给类处理。

对象就是数据，对象本身不包含方法。但是对象有一个“指针”指向一个类，这个类里可以有方法。

方法描述不同属性所导致的不同表现。

类和对象是不可分割的，有对象就必定有一个类和其对应，否则这个对象也就成了没有亲人的孩子（但有一个特殊情况存在，就是由标量进行强制类型转换的object，没有一个类和它对应。此时，PHP中一个称为“孤儿”的stdClass类就会收留这个对象）。

理解了以上四个概念，结合现实世界从实现和存储理解对象和类，这样就不会把二者看成一个抽象、神秘的东西，也就能写出符合现实世界的类了。

如果需要一个类，要从客观世界抽象出一套规律，就得总结这类事物的共性，并且让它可以与其他类进行区分。而这个区分的依据就是属性和方法。区分的办法就是实例化出一个对象，是骡子是马，拉出来遛遛。

现在，你是否对“白马非马”这个典故有了新的认识？

1.2 魔术方法的应用

魔术方法是以两个下画线“”开头、具有特殊作用的一些方法，可以看做PHP的“语法糖”。

语法糖指那些没有给计算机语言添加新功能，而只是对人类来说更“甜蜜”的语法。语法糖往往给程序员提供了更实用的编码方式或者一些技巧性的用法，有益于更好的编码风格，使代码更易读。不过其并没有给语言添加什么新东西。PHP里的引用、SPL等都属于语法糖。

实际上，在1.1节代码中就涉及魔术方法的使用。family类中的construct方法就是一个标准魔术方法。这个魔术方法又称构造方法。具有构造方法的类会在每次创建对象时先调用此方法，所以非常适合在使用对象之前做一些初始化工作。因此，这个方法往往用于类进行初始化时执行一些初始化操作，如给属性赋值、连接数据库等。

以代码清单1-1所示代码为例，family中的construct方法主要做的事情就是在创建对象的同时对属性赋值。也可以这么使用：

```
$tom=new family($student,'peking');
$tom->people->say();
```

这样做就不需要在创建对象后再去赋值了。有构造方法就有对应的析构方法，即destruct方法，析构方法会在某个对象的所有引用都被删除，或者当对象被显式销毁时执行。这两个方法是常见也是最有用的魔术方法。

1.2.1 set和get方法

set和get是两个比较重要的魔术方法，如代码清单1-2所示。

代码清单1-2 magic.php

```
<?php
class Account{
private$user=1;
private$pwd=2;
}
$a=new Account();
echo$a->user;
$a->name=5;
echo$a->name;
```

```
echo$a->big;
```

运行这段代码会怎样呢？结果报错如下：

```
Fatal error: Cannot access private property Account: $user in G: \bak\temp\tempcode\sg.php on line 7
```

所报错误大致是说，不能访问Account对象的私有属性user。在代码清单1-2的类定义里增加以下代码，其中使用了set魔术方法。

```
public function set($name, $value) {
echo"Setting$name to$value\r\n";
$this->$name=$value;
}
public function get($name) {
if(! isset($this->$name)) {
echo'未设置';
$this->$name="正在为你设置默认值";
}
return$this->$name;
}
```

再次运行，看到正常输出，没有报错。在类里以两个下画线开头的方法都属于魔术方法（除非是你自定义的），它们是PHP中的内置方法，有特殊含义。手册里把这两个方法归到重载。

PHP的重载和Java等语言的重载不同。Java里，重载指一个类中可以定义参数列表不同但名字相同的多个方法。比如，Java也有构造函数，Java允许有多个构造函数，只要保证方法签名不一样就行；而PHP则在一个类中只允许有一个构造函数。

PHP提供的“重载”指动态地“创建”类属性和方法。因此，set和get方法被归到重载里。

这里可以直观看到，若类中定义了set和get这一对魔术方法，那么当给对象属性赋值或者取值时，即使这个属性不存在，也不会报错，一定程度上增强了程序的健壮性。

我们注意到，在account类里，user属性的访问权限是私有的，私有属性意味着这个属性是类的“私有财产”，只能在类内部对其进行操作。如果没有set这个魔术方法，直接在类的外部对属性进行赋值操作是会报错的，只能通过在类中定义一个public的方法，然后在类外调用这个公开的方法进行属性读写操作。

现在有了这两个魔术方法，是不是对私有属性的操作变得更方便了呢？实际上，并没有什么奇怪的，因为这两个方法本身就是public的。它们和在对外的public方法中操作private属性的原理一样。只不过这对魔术方法使其操作更简单，不需要显式地调用一个public的方法，因为这对魔术方法在操作类变量时是自动调用的。当然，也可以把类属性定义成public的，这样就可以随意在类的外部进行读写。不过，如果只是为了方便，类属性在任意时候都定义成pub lic权限显然是不合适的，也不符合面向对象的设计思想。

1.2.2 call和callStatic方法

如何防止调用不存在的方法而出错？一样的道理，使用call魔术重载方法。

call方法原型如下：

```
mixed call（string $name, array $arguments）
```

当调用一个不可访问的方法（如未定义，或者不可见）时，call（）会被调用。其中name参数是要调用的方法名称。arguments参数是一个数组，包含着要传递给方法的参数，如下所示：

```
public function call（$name, $arguments） {
switch（count（$arguments））{
case 2:
echo $arguments[0]*$arguments[1], PHP_EOL;
break;
case 3:
echo array_sum（$arguments）, PHP_EOL;
break;
default:
echo' 参数不对' , PHP_EOL;
break;
}
}
$a->make（5）;
$a->make（5, 6）;
```

以上代码模拟了类似其他语言中的根据参数类型进行重载。跟call方法配套的魔术方法是callStatic。当然，使用魔术方法“防止调用不存在的方法而报错”，并不是魔术方法的本意。实际上，魔术方法使方法的动态创建变为可能，这在MVC等框架设计中是很有用的语法。假设一个控制器调用了不存在的方法，那么只要定义了call魔术方法，就能友好地处理这种情况。

试着理解代码清单1-3所示代码。这段代码通过使用callStatic这一魔术方法进行方法的动态创建和延迟绑定，实现一个简单的ORM模型。

代码清单1-3　simpleOrm.php

```
<php
abstract class ActiveRecord {
protected static $table;
protected $fieldvalues;
public $select;
static function findById（$id）{
$query="select*from"
.static: $table
```

```
."where id=$id";
return self::createDomain($query);
}
function get($fieldname) {
return$this->fieldvalues[$fieldname];
}
static function callStatic($method, $args) {
$field=preg_replace('/^findBy(\w*)$/', '${1}', $method);
$query="select*from"
.static::$table
."where$field='$args[0]' ";
return self::createDomain($query);
}
private static function createDomain($query) {
$klass=get_called_class();
$domain=new$klass();
$domain->fieldvalues=array();
$domain->select=$query;
foreach($klass::$fields as$field=>$type) {
$domain->fieldvalues[$field]='TODO: set from sql result';
}
return$domain;
}
}
class Customer extends ActiveRecord{
protected static$table='custdb';
protected static$fields=array(
'id'=>'int',
'email'=>'varchar',
'lastname'=>'varchar'
);
}
class Sales extends ActiveRecord{
protected static$table='salesdb';
protected static$fields=array(
'id'=>'int',
'item'=>'varchar',
'qty'=>'int'
);
}
assert("select*from custdb where id=123"==
Customer::findById(123)->select);
assert("TODO: set from sql result"==
Customer::findById(123)->email);
assert("select*from salesdb where id=321"==
Sales::findById(321)->select);
assert("select*from custdb where Lastname='Denoncourt' "==
Customer::findByLastname('Denoncourt')->select);
```

再举个类似的例子。PHP里有很多字符串函数，假如要先过滤字符串首尾的空格，再求出字符串的长度，一般会这么写：

```
strlen(trim($str));
```

如果要实现JS里的链式操作，比如像下面这样，应该怎么实现？

```
$str->trim()->strlen()
```

很简单，先实现一个String类，对这个类的对象调用方法进行处理时，触发call魔术方法，接着执行call__user__func即可。

1.2.3 toString方法

再看另外一个魔术方法TOstring（在这里故意这么写，是要说明PHP中方法不区分大小写，但实际开发中还需要注意规范）。

当进行测试时，需要知道是否得出正确的数据。比如打印一个对象时，看看这个对象都有哪些属性，其值是什么，如果类定义了toString方法，就能在测试时，echo打印对象体，对象就会自动调用它所属类定义的toString方法，格式化输出这个对象所包含的数据。如果没有这个方法，那么echo一个对象将报错，例如“Catchable fatal error：Object of class Account could not be converted to string”语法错误，实际上这是一个类型匹配失败错误。不过仍然可以用print__r（）和var__dump（）函数输出一个对象。当然，toString是可以定制的，所提供的信息和样式更丰富，如代码清单1-4所示。

代码清单1-4 magic__2.php

```
<？php
class Account｛
public＄user=1；private＄pwd=2；
//自定义的格式化输出方法
public function toString（）｛
return"当前对象的用户名是｛＄this->user｝，密码是｛＄this->pwd｝"；
｝
｝
＄a=new Account（）；echo＄a；
echo PHP__EOL；
print__r（＄a）；
```

运行这段代码发现，使用toString方法后，输出的结果是可定制的，更易于理解。实际上，PHP的toString魔术方法的设计原型来源于Java。Java中也有这么一个方法，而且在Java中，这个方法被大量使用，对于调试程序比较方便。实际上，toString方法也是一种序列化，我们知道PHP自带serialize/unserialize也是进行序列化的，但是这组函数序列化时会产生一些无用信息，如属性字符串长度，造成存储空间的无谓浪费。因此，可以实现自己的序列化和反序列化方法，或者json__encode/json__decode也是一个不错的选择为什么直接echo一个对象就会报语法错误，而如果这个对象实现toString方法后就可以直接输出呢？原因很简单，echo本来可以打印一个对象，而且也实现了这个接口，但是PHP对其做了个限制，只有实现toString后才允许使用。从下面的PHP源代码里可以得到验证：

```
ZEND__VM__HANDLER（40，ZEND__ECHO，CONST｜TMP｜VAR｜CV，ANY）
｛
zend__op*opline=EX（opline）；
zend__free__op free__op1；
zval z__copy；
zval*z=GET__OP1__ZVAL__PTR（BP__VAR__R）；
//此处的代码预留了把对象转换为字符串的接口
if（OP1__TYPE！=IS__CONST&&
Z__TYPE__P（z）==IS__OBJECT&&Z__OBJ__HT__P（z）->get__method！=NULL&&
zend__std__cast__object__tostring（z，&z__copy，IS__STRING TSRMLS__CC）==SUCCESS）｛
zend__print__variable（&z__copy）；
zval__dtor（&z__copy）；
｝else｛
zend__print__variable（z）；
｝
FREE__OP1（）；
ZEND__VM__NEXT__OPCODE（）；
｝
```

由此可见，魔术方法并不神奇。

有比较才有认知。最后，针对本节代码给出一个Java版本的代码，供各位读者用来对比两种语言中重载和魔术方法的异同，如代码清单1-5所示。

代码清单1-5　Account.java

```
import org.apache.commons.lang3.builder.ToStringBuilder；
/**
*类的重载演示Java版本
*@author wfox
*@date@verson
*/
public class Account｛
private String user；//用户名
private String pwd；//密码
public Account（）｛
System.out.println（"构造函数"）；
｝
public Account（String user，String pwd）｛
System.out.println（"重载构造函数"）；
System.out.println（user+"——"+pwd）；
｝
public void say（String user）｛
System.out.println（"用户是："+user）；
｝
public void say（String user，String pwd）｛
System.out.println（"用户："+user）；
System.out.println（"密码"+pwd）；
｝
public String getUser（）｛
return user；
｝
public void setUser（String user）｛
this.user=user；
｝
public String getPwd（）｛
return pwd；
｝
public void setPwd（String pwd）｛
｝
@Override
public String toString（）｛
return ToStringBuilder.reflectionToString（this）；
｝
public static void main（String……）｛
Account account=new Account（）；
account.setUser（"张三"）；
account.setPwd（"123456"）；
account.say（"李四"）；
account.say（"王五"，"123"）；
System.out.println（account）；
｝
｝
```

运行上述代码，输出如图1-2所示。

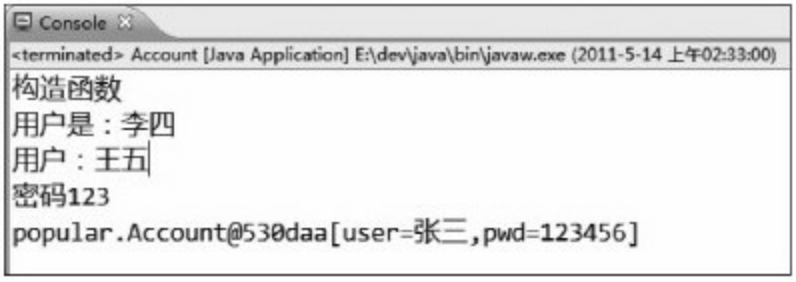

图　1-2　Java里的构造方法和重载演示

可以看出，Java的构造方法比PHP好用，PHP由于有了set/get这一对魔术方法，使得动态增加对象的属性字段变得很方便，而对Java来说，要实现类似的效果，就不得不借助反射API或直接修改编译后字节码的方式来实现。这体现了动态语言的优势，简单、灵活。

1.3　继承与多态

面向对象的优势在于类的复用。继承与多态都是对类进行复用，它们一个是类级别的复用，一个是方法级别的复用。提到继承必提组合，二者有何异同？PHP到底有没有多态？若没有，则为什么没有？有的话，和其他语言中的多态又有什么区别？这些都是本节所要讲述的内容。

1.3.1　类的组合与继承

在1.1节的代码中定义了两个类，一个是person，一个是family；在family类中创建person类中的对象，把这个对象视为family类的一个属性，并调用它的方法处理问题，这种复用方式叫“组合”。还有一种复用方式，就是继承。

类与类之间有一种父与子的关系，子类继承父类的属性和方法，称为继承。在继承里，子类拥有父类的方法和属性，同时子类也可以有自己的方法和属性。

可以把1.1节的组合用继承实现，如代码清单1-6所示。

代码清单1-6　family__extends.php

```
<? php
class person {
public $name=' Tom' ; public $gender;
static $money=10000;
public function construct () {
echo' 这里是父类' , PHP__EOL;
}
public function say () {
echo $this->name, "\tis", $this->gender, "\r\n";
}
}
class family extends person {
public $name; public $gender; public $age;
static $money=100000;
public function construct () {
parent: construct () ; //调用父类构造方法
echo' 这里是子类' , PHP__EOL;
}
public function say () {
parent: say () ;
echo $this->name, "\tis\t", $this->gender, ", and
is\t", $this->age, PHP__EOL;
}
public function cry () {
echo parent: $money, PHP__EOL;
echo' %>__<%' , PHP__EOL;
echo self: $money, PHP__EOL; //调用自身构造方法
echo' (*^ ^*) ' ;
}
}
$poor=new family () ;
$poor->name=' Lee' ;
$poor->gender=' female' ;
```

```
$poor->age=25;
$poor->say();
$poor->cry();
```

运行上面的代码，可以得到如下输出结果：

```
这里是父类
这里是子类
Lee  is female
Lee  is  female, and is  25
10000
%>__<%
100000
（*^ ^*）
```

从上面代码中可以了解继承的实现。在继承中，用parent指代父类，用self指代自身。使用“：”运算符（范围解析操作符）调用父类的方法。“：”操作符还用来作为类常量和静态方法的调用，不要把这两种应用混淆。

既然提到静态，就再强调一点，如果声明类成员或方法为static，就可以不实例化类而直接访问，同时也就不能通过一个对象访问其中的静态成员（静态方法除外），也不能用“：”访问一个非静态方法。比如，把上例中的poor->cry（）；换成poor：cry（），按照这个规则，应该是要报错的。可能试验时，并没有报错，而且能够正确输出。这是因为用“：”方式调用一个非静态方法会导致一个E__STRICT级别的错误，而这里的PHP设置默认没有开启这个级别的报错提示。打开PHP安装目录下的php.ini文件，设置如下：

```
error__reporting=E__ALL|E__STRICT display__errors=On
```

再次运行，就会看到错误提示。因此，用“：”访问一个非静态方法不符合语法，但PHP仍然能够正确地执行代码，这只是PHP所做的一个“兼容”或者说“让步”。在开发时，设置最严格的报错等级，在部署时可适当调低。

组合与继承都是提高代码可重用性的手段。在设计对象模型时，可以按照语义识别类之间的组合关系和继承关系。比如，通过一些总结，得出了继承是一种“是、像”的关系，而组合是一种“需要”的关系。利用这条规律，就可以很简单地判断出父亲与儿子应该是继承关系，父亲与家庭应该是组合关系。还可以从另外一个角度看，组合偏重整体与局部的关系，而继承偏重父与子的关系，如图1-3所示。

图　1-3　继承和组合的对照

从方法复用角度考虑，如果两个类具有很多相同的代码和方法，可以从这两个类中抽象出一个父类，提供公共方法，然后两个类作为子类，提供个性方法。这时用继承语意更好。继承的UML图如图1-4所示。

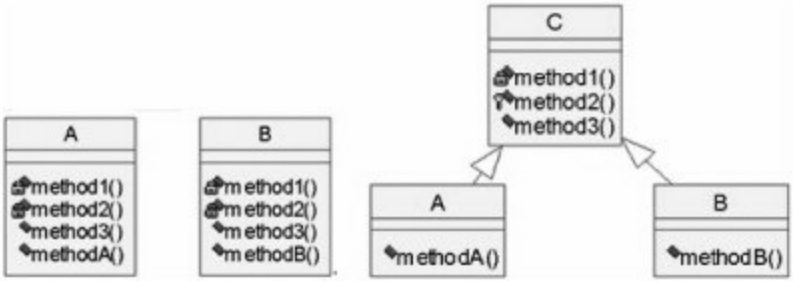

图　1-4　继承的UML图

而组合就没有这么多限制。组合之间的类可以关系（体现为复用代码）很小，甚至没有关系，如图1-5所示。

图　1-5　组合的UML图

然而在编程中，继承与组合的取舍往往并不是这么直接明了，很难说出二者是“像”的关系还是“需要”的关系，甚至把它拿到现实世界中建模，还是无法决定应该是继承还是组合。那应该怎么办呢？有什么标准吗？有的。这个标准就是“低耦合”。

耦合是一个软件结构内不同模块之间互连程度的度量，也就是不同模块之间的依赖关系。

低耦合指模块与模块之间，尽可能地使模块间独立存在；模块与模

块之间的接口尽量少而简单。现代的面向对象的思想不强调为真实世界建模，变得更加理性化一些，把目标放在解耦上。

解耦是要解除模块与模块之间的依赖。

按照这个思想，继承与组合二者语义上难于区分，在二者均可使用的情况下，更倾向于使用组合。为什么呢？继承存在什么问题呢？

1）继承破坏封装性。

比如，定义鸟类为父类，具有羽毛属性和飞翔方法，其子类天鹅、鸭子、鸵鸟等继承鸟这个类。显然，鸭子和鸵鸟不需要飞翔这个方法，但作为子类，它们却可以无区别地使用飞翔这个方法，显然破坏了类的封装性。而组合，从语义上来说，要优于继承。

2）继承是紧耦合的。

继承使得子类和父类捆绑在一起。组合仅通过唯一接口和外部进行通信，耦合度低于继承。

3）继承扩展复杂。

随着继承层数的增加和子类的增加，将涉及大量方法重写。使用组合，可以根据类型约束，实现动态组合，减少代码。

4）不恰当地使用继承可能违反现实世界中的逻辑。

比如，人作为父类，雇员、经理、学生作为子类，可能存在这样的问题，经理一定是雇员，学生也可能是雇员，而使用继承的话一个人就无法拥有多个角色。这种问题归结起来就是“角色”和“权限”问题。在权限系统中很可能存在这样的问题，经理权利和职位大于主管，但出于分工和安全的考虑，经理没有权限直接操作主管所负责的资源，技术部经理也没权限直接命令市场部主管。这就要求角色和权限系统的设计要更灵活。不恰当的继承可能导致逻辑混乱，而使用组合就可以较好地解决这个问题。

当然，组合并非没有缺点。在创建组合对象时，组合需要一一创建局部对象，这一定程度上增加了一些代码，而继承则不需要这一步，因为子类自动有了父类的方法，如代码清单1-7所示。

代码清单1-7　mobile.php

```
<? php
//继承拥有比组合更少的代码量
class car {
public function addoil () {
echo"Add oil\r\n";
}
}
class bmw extends car {
}
class benz {
public $car;
public function construct () {
$this->car=new car;
}
public function addoil () {
$this->car->addoil ();
}
}
$bmw=new bmw;
$bmw->addoil ();
$benz=new benz ();
$benz->addoil ();
```

显然，组合比继承增加了代码量。组合还有其他的一些缺点，不过总体说来，是优点大于缺点。

继承最大的优点就是扩展简单，但是其缺点大于优点，所以在设计时，需要慎重考虑。那应该如何使用继承呢？

精心设计专门用于被继承的类，继承树的抽象层应该比较稳定，一般不要多于三层。

对于不是专门用于被继承的类，禁止其被继承，也就是使用final修饰符。使用final修饰符既可防止重要方法被非法覆写，又能给编辑器寻找优化的机会。

优先考虑用组合关系提高代码的可重用性。

子类是一种特殊的类型，而不只是父类的一个角色。

子类扩展，而不是覆盖或者使父类的功能失效。

底层代码多用组合，顶层/业务层代码多用继承。底层用组合可以提高效率，避免对象臃肿。顶层代码用继承可以提高灵活性，让业务使用更方便。

思考题　设计一个log类，需要用到MySQL中的CURD操作，是应该使用继承呢还是组合？请给出理由。

继承并非一无是处，而组合也不是完美无缺的。如果既要组合的灵活，又要继承的代码简洁，能做到吗？

这是可以做到的，譬如多重继承，就具有这个特性。多重继承里一个类可以同时继承多个父类，组合两个父类的功能。C++里就是使用的这种模型来增强继承的灵活性的，但是多重继承过于灵活，并且会带来“菱形问题”，故为其使用带来了不少困难，模型变得复杂起来，因此在大多数语言中，都放弃了多重继承这一模型。

多重继承太复杂，那么还有其他方式能比较好地解决这个问题吗？PHP5.4引入的新的语法结构Traits就是一种很好的解决方案。Traits的思想来源于C++和Ruby里的Mixin以及Scala里的Traits，可以方便我们实现对象的扩展，是除extend、implements外的另外一种扩展对象的方式。Traits既可以使单继承模式的语言获得多重继承的灵活，又可以避免多重继承带来的种种问题。

1.3.2 各种语言中的多态

多态确切的含义是：同一类的对象收到相同消息时，会得到不同的结果。而这个消息是不可预测的。多态，顾名思义，就是多种状态，也就是多种结果。

以Java为例，由于Java是强类型语言，因此变量和函数返回值是有状态的。比如，实现一个add函数的功能，其参数可能是两个int型整数，也可能是两个float型浮点数，而返回值可能是整型或者浮点型。在这种情况下，add函数是有状态的，它有多种可能的运行结果。在实际使用时，编译器会自动匹配适合的那个函数。这属于函数重载的概念。需要说明的是，重载并不是面向对象里的东西，和多态也不是一个概念，它属于多态的一种表现形式。

多态性是一种通过多种状态或阶段描述相同对象的编程方式。它的真正意义在于：实际开发中，只要关心一个接口或基类的编程，而不必关心一个对象所属于的具体类。

很多地方会看到“PHP没有多态”这种说法。事实上，不是它没有，而是它本来就是多态的。PHP作为一门脚本语言，自身就是多态的，所以在语言这个级别上，不谈PHP的多态。在PHP官方手册也找不到对多态的详细描述。

既然说PHP没有多态这个概念（实际上是不需要多态这个概念），那为什么又要讲多态呢？可以看下面的例子，如代码清单1-8所示。

代码清单1-8　Polymorphism.php

```
<? php
class employee {
protected function working () {
echo' 本方法需重载才能运行' ;
}
}
class teacher extends employee {
public function working () {
echo' 教书' ;
}
}
class coder extends employee {
public function working () {
echo' 敲代码' ;
}
}
function doprint ( $obj) {
if (get_class ( $obj) ==' employee' ) {
echo' Error' ;
} else {
$obj->working () ;
```

```
}
}
doprint（new teacher（））；
doprint（new coder（））；
doprint（new employee（））；
```

通过判断传入的对象所属的类不同来调用其同名方法，得出不同结果，这是多态吗？如果站在C++角度，这不是多态，这只是不同类对象的不同表现而已。C++里的多态指运行时对象的具体化，指同一类对象调用相同的方法而返回不同的结果。看个C++的例子，如代码清单1-9所示。

代码清单1-9　C++多态的例子

```
#include＜cstdlib＞
#include＜iostream＞
/**
C++中用虚函数实现多态
*/
using namespace std；
class father｛
public：
father（）：age（30）｛cout＜＜"父类构造法，年龄"＜＜age＜＜"\n"；｝
～father（）｛cout＜＜"父类析构"＜＜"\n"；｝
void eat（）｛cout＜＜"父类吃饭吃三斤"＜＜"\n"；｝
virtual void run（）｛cout＜＜"父类跑10000米"＜＜"\n"；｝//虚函数
protected：
int age；
｝；
class son：public father｛
public：
son（）｛cout＜＜"子类构造法"＜＜"\n"；｝
～son（）｛cout＜＜"子类析构"＜＜"\n"；｝
void eat（）｛cout＜＜"儿子吃饭吃一斤"＜＜"\n"；｝
void run（）｛cout＜＜"儿子跑100米"＜＜"\n"；｝
void cry（）｛cout＜＜"哭泣"＜＜"\n"；｝
｝；
int main（int argc, char*argv[]）
｛
father*pf=new son；
pf->eat（）；
pf->run（）；
delete pf；
system（"PAUSE"）；
return EXIT_SUCCESS；
｝
```

上面的代码首先定义一个父类，然后定义一个子类，这个子类继承父类的一些方法并且有自己的方法。通过father*pf=new　son；语句创建一个派生类（子类）对象，并且把该派生类对象赋给基类（父类）指针，然后用该指针访问父类中的eat和run方法。图1-6所示是运行结果。

由于父类中的run方法加了virtual关键字，表示该函数有多种形态，可能被多个对象所拥有。也就是说，多个对象在调用同一名字的函数时会产生不同的效果。

这个例子和PHP的例子有什么不同呢？C++的这个例子所创建的对象是一个指向父类的子对象，还可以创建更多派生类对象，然后上转型为父类对象。这些对象，都是同一类对象，但是在运行时，却都能调用

到派生类同名函数。而PHP中的例子则是不同类的对象调用。

图　1-6　运行结果

由于PHP是弱类型的，并且也没有对象转型机制，所以不能像C++或者Java那样实现father＄pf=new　　son；把派生类对象赋给基类对象，然后在调用函数时动态改变其指向。在PHP的例子中，对象都是确定的，是不同类的对象。所以，从这个角度讲，这还不是真正的多态。

代码清单1-8所示代码通过判断对象的类属性实现“多态”，此外，还可以通过接口实现多态，如代码清单1-10所示。

代码清单1-10　通过接口实现多态

```
<? php
interface employee {
public function working () ;
}
class teacher implements employee {
public function working () {
echo' 教书' ;
}
}
class coder implements employee {
public function working () {
echo' 敲代码' ;
}
}
function doprint (employee $i) {
$i  $i->working () ;
}
$a=new teacher;
$b=new coder;
doprint ( $a) ;
doprint ( $b) ;
```

这是多态吗？这段代码和代码清单1-8相比没有多少区别，不过这段代码中doprint函数的参数是一个接口类型的变量，符合“同一类型，不同结果”这一条件，具有多态性的一般特征。因此，这是多态。

如果把代码清单1-8中doprint函数的obj参数看做一种类型（把所有弱类型看做一种类型），那就可以认为代码清单1-8中的代码也是一种多态。

再次把三段代码放在一起品味，可以看出：区别是否是多态的关键

在于看对象是否属于同一类型。如果把它们看做同一种类型，调用相同的函数，返回了不同的结果，那么它就是多态；否则，不能称其为多态。由此可见，弱类型的PHP里多态和传统强类型语言里的多态在实现和概念上是有一些区别的，而且弱类型语言实现起多态来会更简单，更灵活。

本节解决了什么是多态，什么不是多态的问题。至于多态是怎么实现的，各种语言的策略是不一样的。但是，最终的实现无非就是查表和判断。总结如下：

多态指同一类对象在运行时的具体化。

PHP语言是弱类型的，实现多态更简单、更灵活。

类型转换不是多态。

PHP中父类和子类看做“继父”和“继子”关系，它们存在继承关系，但不存在血缘关系。因此子类无法向上转型为父类，从而失去多态最典型的特征。

多态的本质就是if……else，只不过实现的层级不同。

1.4　面向接口编程

这里，首先强调一个概念，面向接口编程并不是一种新的编程范式。本章开头提到的三大范式中并没有提到面向接口。其次，这里是狭义的接口，即interface关键字。广义的接口可以是任何一个对外提供服务的出口，比如提供数据传输的USB接口、淘宝网对其他网站开放的支付宝接口。

1.4.1　接口的作用

接口定义一套规范，描述一个“物”的功能，要求如果现实中的“物”想成为可用，就必须实现这些基本功能。接口这样描述自己：

“对于实现我的所有类，看起来都应该像我现在这个样子”。

采用一个特定接口的所有代码都知道对于那个接口会调用什么方法。这便是接口的全部含义。接口常用来作为类与类之间的一个“协议”。接口是抽象类的变体，接口中所有方法都是抽象的，没有一个有程序体。接口除了可以包含方法外，还能包含常量。

比如用接口描述发动机，要求机动车必须要有“run”功能，至于怎么实现（摩托还是宝马），应该是什么样（前驱还是后驱），不是接口关心的。因为接口为抽象而生。作为质检总局，要判断这辆车是否合格，只要按“接口”的定义一条一条验证，这辆车不能“run”，那它就是废品，不能通过验收。但是，如果汽车实现了接口中本来不存在的方法music，并不认为有什么问题。接口就是一种契约。因此，在程序里，接口的方法必须被全部实现，否则将报fetal错误，如代码清单1-11所示。

代码清单1-11　interface.php

```
<? php
interface mobile
 {
public function run（）；//驱动方法
}
class plain implements mobile
 {
public function run（）
 {
echo"我是飞机"；
}
public function fly（）
 {
```

```
echo"飞行";
}
}
class car implements mobile{
public function run() {
echo"我是汽车\r\n";
}
}
class machine
 {
function demo(mobile$a)
 {
$a->fly(); //mobile接口是没有这个方法的
}
}
$obj=new machine();
$obj->demo(new plain()); //运行成功
$obj->demo(new car()); //运行失败
```

在这段代码里，定义一个机动车接口，其中含有一个发动机功能。然后用一个飞机类实现这个接口，并增加了飞行方法。最后，在一个机械检测类中对机动车进行测试（用类型约束指定要测试的是机动车这个接口）。但是，此检测线测试的却是机动车接口中不存在的fly方法，直到遇到car的实例因不存在fly方法而报错。

这段代码实际上是错误的，不符合接口语义。但是在PHP里，对plain的实例进行检测时是可以运行的。也就是说，在PHP里，只关心是否实现这个方法，而并不关心接口语义是否正确。

按理说，接口应该是起一个强制规范和契约的作用，但是这里对接口的约束并没有起效，也打破了契约，对检测站这个类的行为失去控制。可以看看在Java里是怎么处理的，如图1-7所示。

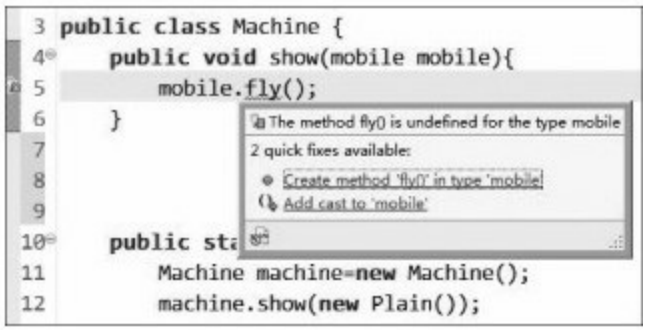

图　1-7　Java中接口是一种类型

Java认为，接口就是一种type，即类型。如果你打破了我们之间的契约，你的行为变得无法控制，那就是非法的。这符合逻辑，也符合现实世界。这就真正起到接口作为规范的作用了。

接口不仅规范接口的实现者，还规范接口的执行者，不允许调用接口中本不存在的方法。当然这并不是说一个类如果实现了接口，就只能

实现接口中才有的方法，而是说，如果针对的是接口，而不是具体的类，则只能按接口的约定办事。这样的语法规定对接口的使用是有利的，让程序更健壮。根据这个角度讲，为了保证接口的语义，通常一个接口的实现类仅实现该接口所具有的方法，做到专一，当然这也不是一成不变的。

由上面的例子可以看出，PHP里接口作为规范和契约的作用打了折扣。上面例子实际就是一个典型面向接口编程的例子。根据这个例子，可以很自然地想到使用接口的场合，比如数据库操作、缓存实现等。不用关心我们所面对的数据库是MySQL还是Oracle，只需要关心面向Database接口进行具体业务的逻辑相关的代码，这就是面向接口编程的来历。

在这里，Database就如同employee一样，针对这个接口实现就好了。缓存功能也一样，我们不关注缓存是内存缓存还是文件缓存，或者是数据库缓存，只关注它是否实现了Cache接口，并且它只要实现了Cache接口，就实现了写入缓存和读取某条缓存中的数据及清除缓存这几个关键的功能点。

通常在大型项目里，会把代码进行分层和分工。核心开发人员和技术经理编写核心的流程和代码，往往是以接口的形式给出，而基础开发人员则针对这些接口，填充代码，如数据库操作等。这样，核心技术人员把更多精力投入到了技术攻关和业务逻辑中。前端针对接口编程，只管在Action层调用Service，不管实现细节；而后端则要负责Dao和Service层接口实现。这样，就实现了代码分工与合作。

1.4.2　对PHP接口的思考

PHP的接口自始至终一直在被争议，有人说接口很好，有人说接口像鸡肋。首先要明白，好和不好的判断标准是什么。无疑，这是和Java/C++相比。在上面的例子中，已经讨论了PHP的接口在“面向契约编程”中是不足的，并没有起到应有的作用。

其实，在上面的interface.php代码中，machine类的声明应该在plain类前面。接口提供了一套规范，这是系统提供的，然后machine类提供一组针对接口的API并实现，最后才是自定义的类。在Java里，接口之所以盛行（多线程的runable接口、容器的collection接口等）就是因为系统为我们做了前面两部分的工作，而程序员，只需要去写具体的实现类，就能保证接口可用可控。

为什么要用接口？接口到底有什么好处？接口本身并不提供实现，只是提供一个规范。如果我们知道一个类实现了某个接口，那么就知道了可以调用该接口的哪些方法，我们只需要知道这些就够了。

PHP中，接口的语义是有限的，使用接口的地方并不多，PHP中接口可以淡化为设计文档，起到一个团队基本契约的作用，代码清单1-12所示。

代码清单1-12　cache__imp.php

```
<? php
interface cache {
/**
@describe：缓存管理，项目经理定义接口，技术人员负责实现
**/
const maxKey=10000; //最大缓存量
public function getc（$key）; //获取缓存
public function setc（$key, $value）; //设置缓存
public function flush（）; //清空缓存
}
```

由于PHP是弱类型，且强调灵活，所以并不推荐大规模使用接口，而是仅在部分“内核”代码中使用接口，因为PHP中的接口已经失去很多接口应该具有的语义。从语义上考虑，可以更多地使用抽象类。至于抽象类和接口的比较，不再赘述。

另外，PHP5对面向对象的特性做了许多增强，其中就有一个SPL（标准PHP库）的尝试。SPL中实现一些接口，其中最主要的就是

Iterator迭代器接口，通过实现这个接口，就能使对象能够用于foreach结构，从而在使用形式上比较统一。比如SPL中有一个DirectoryIterator类，这个类在继承SplFileInfo类的同时，实现Iterator、Traversable、SeekableIterator这三个接口，那么这个类的实例可以获得父类SplFileInfo的全部功能外，还能够实现Iterator接口所展示的那些操作。

Iterator接口的原型如下：

```
*current()
This method returns the current index's value.You are solely
responsible for tracking what the current index is as the
interface does not do this for you.
*key()
This method returns the value of the current index's key.For
foreach loops this is extremely important so that the key
value can be populated.
*next()
This method moves the internal index forward one entry.
*rewind()
This method should reset the internal index to the first element.
*valid()
This method should return true or false if there is a current
element.It is called after rewind() or next().
```

如果一个类声明了实现Iterator接口，就必须实现这五个方法，如果实现了这五个方法，那么就可以很容易对这个类的实例进行迭代。这里，DirectoryIterator类之所以拿来就能用，是因为系统已经实现了Iterator接口，所以可以像下面这样使用：

```
<? php
$dir=new DirectoryIterator(dirname(FILE));
foreach($dir as$fileinfo){
if(! $fileinfo->isDir()){
echo
$fileinfo->getFilename(), "\t", $fileinfo->getSize(), PHP_EOL;
}
}
```

可以想象，如果不用DirectoryIterator类，而是自己实现，不但代码量增加了，而且循环时候的风格也不统一了。如果自己写的类也实现了Iterator接口，那么就可以像Iterator那样工作。

为什么一个类只要实现了Iterator迭代器，其对象就可以被用作foreach的对象呢？其实原因很简单，在对PHP实例对象使用foreach语法时，会检查这个实例有没有实现Iterator接口，如果实现了，就会通过内置方法或使用实现类中的方法模拟foreach语句。这是不是和前面提到的toString方法的实现很像呢？事实上，toString方法就是接口的一种变相实现。

接口就是这样，接口本身什么也不做，系统悄悄地在内部实现了接

口的行为，所以只要实现这个接口，就可以使用接口提供的方法。这就是接口“即插即用”思想。

我们都知道，接口是对多重继承的一种变相实现，而在讲继承时，我们提到了用来实现混入（Mixin）式的Traits，实际上，Traits可以被视为一种加强型的接口。

来看一段代码：

```
<? php
trait Hello {
public function sayHello () {
echo' Hello' ;
}
}
trait World {
public function sayWorld ()
echo' World' ;
}
}
class MyHelloWorld {
use Hello, World;
public function sayExclamationMark () {
echo' ! ' ;
}
}
$o=new MyHelloWorld () ;
$o->sayHello () ;
$o->sayWorld () ;
$o->sayExclamationMark () ;
? >
```

上面的代码运行结果如下：

```
Hello World!
```

这里的MyHelloWorld同时实现了两个Traits，从而使其可以分别调用两个Traits里的代码段。从代码中就可以看出，Traits和接口很像，不同的是Traits是可以导入包含代码的接口。从某种意义上来说，Traits和接口都是对“多重继承”的一种变相实现。

总结关于接口的几个概念：

接口作为一种规范和契约存在。作为规范，接口应该保证可用性；作为契约，接口应该保证可控性。

接口只是一个声明，一旦使用interface关键字，就应该实现它。可以由程序员实现（外部接口），也可以由系统实现（内部接口）。接口本身什么都不做，但是它可以告诉我们它能做什么。

PHP中的接口存在两个不足，一是没有契约限制，二是缺少足够多的内部接口。

接口其实很简单，但是接口的各种应用很灵活，设计模式中也有很大一部分是围绕接口展开的。

1.5 反射

面向对象编程中对象被赋予了自省的能力，而这个自省的过程就是反射。

反射，直观理解就是根据到达地找到出发地和来源。比方说，我给你一个光秃秃的对象，我可以仅仅通过这个对象就能知道它所属的类、拥有哪些方法。

反射指在PHP运行状态中，扩展分析PHP程序，导出或提取出关于类、方法、属性、参数等的详细信息，包括注释。这种动态获取信息以及动态调用对象方法的功能称为反射API。

1.5.1 如何使用反射API

以1.1节的代码为模板，直观地认识反射的使用，如代码清单1-13所示。

代码清单1-13 reflection.php

```
<? php
class person {
public $name;
public $gender;
public function say() {
echo $this->name, "\tis", $this->gender, "\r\n";
}
public function set($name, $value) {
echo"Setting $name to $value\r\n";
$this->$name=$value;
}
public function get($name) {
if(! isset($this->$name)) {
echo' 未设置';
$this->$name="正在为你设置默认值";
}
return $this->$name;
} }
$student=new person();
$student->name=' Tom';
$student->gender=' male';
$student->age=24;
```

现在，要获取这个student对象的方法和属性列表该怎么做呢？如以下代码所示：

```
//获取对象属性列表
$reflect=new ReflectionObject($student);
$props =$reflect->getProperties();
foreach($props as$prop) {
print$prop->getName()."\n";
}
```

```
//获取对象方法列表
$m=$reflect->getMethods();
foreach($m as$prop) {
print$prop->getName()."\n";
}
```

也可以不用反射API，使用class函数，返回对象属性的关联数组以及更多的信息：

```
//返回对象属性的关联数组
var_dump(get_object_vars($student));
//类属性
var_dump(get_class_vars(get_class($student)));
//返回由类的方法名组成的数组
var_dump(get_class_methods(get_class($student)));
```

假如这个对象是从其他页面传过来的，怎么知道它属于哪个类呢？一句代码就可以搞定：

```
//获取对象属性列表所属的类
echo get_class($student);
```

反射API的功能显然更强大，甚至能还原这个类的原型，包括方法的访问权限，如代码清单1-14所示。

代码清单1-14　使用反射API

```
//反射获取类的原型
$obj=new ReflectionClass('person');
$className=$obj->getName();
$Methods=$Properties=array();
foreach($obj->getProperties()as$v)
{
$Properties[$v->getName()]=$v;
}
foreach($obj->getMethods()as$v)
{
$Methods[$v->getName()]=$v;
}
echo"class{$className}\n{\n";
is_array($Properties)&&ksort($Properties);
foreach($Properties as$k=>$v)
{
echo"\t";
echo$v->isPublic()?'public':'',$v->isPrivate()?'private':'',
$v->isProtected()?'protected':'',
$v->isStatic()?'static':'';
echo"\t{$k}\n";
}
echo"\n";
if(is_array($Methods))ksort($Methods);
foreach($Methods as$k=>$v)
{
echo"\tfunction{$k}(){}\n";
}
echo"}\n";
```

输出如下：

```
class person
```

```
{
public  gender
public  name
function get() {}
function set() {}
function say() {}
}
```

不仅如此，PHP手册中关于反射API更是有几十个，可以说，反射完整地描述了一个类或者对象的原型。反射不仅可以用于类和对象，还可以用于函数、扩展模块、异常等。

1.5.2 反射有什么作用

反射可以用于文档生成。因此可以用它对文件里的类进行扫描，逐个生成描述文档。

既然反射可以探知类的内部结构，那么是不是可以用它做hook实现插件功能呢？或者是做动态代理呢？抛砖引玉，代码清单1-15是个简单的举例。

代码清单1-15　proxy.php

```
<? php
class mysql {
function connect（$db） {
echo"连接到数据库$ {db[0]} \r\n";
}
}
class sqlproxy {
private$target;
function construct（$tar） {
$this->target[]=new$tar（）;
}
function call（$name, $args） {
foreach（$this->target as$obj） {
$r=new ReflectionClass（$obj）;
if（$method=$r->getMethod（$name）） {
if（$method->isPublic（） &&! $method->isAbstract（）） {
echo"方法前拦截记录LOG\r\n";
$method->invoke（$obj, $args）;
echo"方法后拦截\r\n";
}
}
}
}
}
$obj=new sqlproxy（' mysql' ）;
$obj->connect（' member' ）;
```

这里简单说明一下，真正的操作类是mysql类，但是sqlproxy类实现了根据动态传入参数，代替实际的类运行，并且在方法运行前后进行拦截，并且动态地改变类中的方法和属性。这就是简单的动态代理。

在平常开发中，用到反射的地方不多：一个是对对象进行调试，另一个是获取类的信息。在MVC和插件开发中，使用反射很常见，但是反射的消耗也很大，在可以找到替代方案的情况下，就不要滥用。

PHP有Token函数，可以通过这个机制实现一些反射功能。从简单灵活的角度讲，使用已经提供的反射API是可取的。

很多时候，善用反射能保持代码的优雅和简洁，但反射也会破坏类的封装性，因为反射可以使本不应该暴露的方法或属性被强制暴露了出

来，这既是优点也是缺点。

思考题　为什么要使用反射，反射存在的必要性是什么？或者说，反射为什么会存在？（已知一些情况：C语言是面向过程的编程语言，PHP、C++、Java是具有面向对象风格的编程语言。C语言和C++中没有对反射的原生支持，而PHP和Java具有反射API。可以思考一下，为什么C/C++语言里没有反射，以及C/C++语言里是否需要反射？）

1.6 异常和错误处理

在语言级别上，通常具有许多错误处理模式，但这些模式往往建立在约定俗成的基础上，也就是说这些错误都是预知的。但是在大型程序中，如果每次调用都去逐一检查错误，会使代码变得冗长复杂，到处充斥着if……else，并且严重降低代码的可读性。而且人的因素也是不可信赖的，程序员可能并不会把这些问题当一回事，从而导致业务异常。在这种背景下，就逐渐形成了异常处理机制，或者强迫消除这些问题，或者把问题提交给能解决它的环境。这就把“描述在正常过程中做什么事的代码”和“出了问题怎么办的代码”进行分离。

1.6.1 如何使用异常处理机制

异常的思想最早可以追溯到20世纪60年代，其在C++、Java中发扬光大，PHP则部分借鉴了这两种语言的异常处理机制。

PHP里的异常，是程序运行中不符合预期的情况及与正常流程不同的状况。一种不正常的情况，就是按照正常逻辑不该出错，但仍然出错的情况，这属于逻辑和业务流程的一种中断，而不是语法错误。PHP里的错误则属于自身问题，是一种非法语法或者环境问题导致的、让编译器无法通过检查甚至无法运行的情况。

在各种语言里，异常（exception）和错误（error）的概念是不一样的。在PHP里，遇到任何自身错误都会触发一个错误，而不是抛出异常（对于一些情况，会同时抛出异常和错误）。PHP一旦遇到非正常代码，通常都会触发错误，而不是抛出异常。在这个意义上，如果想使用异常处理不可预料的问题，是办不到的。比如，想在文件不存在且数据库连接打不开时触发异常，是不可行的。这在PHP里把它作为错误抛出，而不会作为异常自动捕获。

以经典的除零问题为例，如代码清单1-16所示。

代码清单1-16　exception.php

```
//exception.php
<? php
$a=null;
try {
$a=5/0;
```

```
echo $a, PHP_EOL;
} catch (exception $e) {
$e->getMessage();
$a=-1;
}
echo $a;
```

运行结果如图1-8所示。

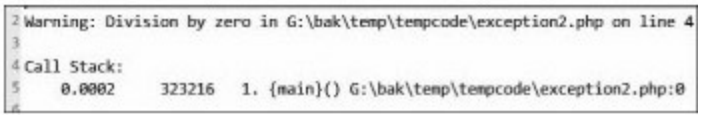

图 1-8 PHP里的除零错误

代码清单1-17所示是Java代码。

代码清单1-17 ExceptionTry.java

```
//ExceptionTry.java
public class ExcepetionTry {
public static void tp() throws ArithmeticException {
int a;
a=5/0;
System.out.println("运算结果："+a);
}
public static void main(String[]args) {
int a; try {
a=5/0;
System.out.println("运算结果："+a);
} catch (ArithmeticException e) {
e.printStackTrace(); } finally {
a=-1;
System.out.println("运算结果："+a);
}
try {
ExcepetionTry.tp();
} catch (Exception e) {
System.out.println("异常被捕获");
}
}
}
```

运行结果如图1-9所示。

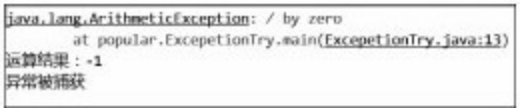

图 1-9 Java里的除零异常

把tp方法中的第二条语句改为如下形式：

```
a=5/1;
```

修改后的结果如图1-10所示。

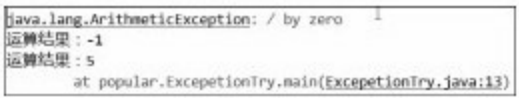

图　1-10　Java里的异常

由以上运行结果可以看到，对于除零这种“异常”情况，PHP认为这是一个错误，直接触发错误（warning也是错误，只是错误等级不一样），而不会自动抛出异常使程序进入异常流程，故最终a值并不是预想中的-1，也就是说，并没有进入异常分支，也没有处理异常。PHP只有你主动throw后，才能捕获异常（一般情况是这样，也有一些异常PHP可以自动捕获）。

而对于Java，则认为除零属于ArithmeticException，会对其进行捕获，并对异常进行处理。

也就是说，PHP通常是无法自动捕获有意义的异常的，它把所有不正常的情况都视作了错误，你要想捕获这个异常，就得使用if……else结构，保证代码是正常的，然后判断如果除数为0，则手动抛出异常，再捕获。Java有一套完整的异常机制，内置很多异常类会自动捕获各种各样的异常。但PHP这个机制不完善。PHP内建的常见异常类主要有pdoexception、reflection exception。

注意　其实PHP和Java之间之所以有这个差距，根本原因就在于，在Java里，异常是唯一的错误报告方式，而在PHP中却不是这样。通俗一点讲，就是这两种语言对异常和错误的界定存在分歧。什么是异常，什么是错误，两种语言的设计者存在不同的观点。

也就是说，PHP只有手动抛出异常后才能捕获异常，或者是有内建的异常机制时，会先触发错误，再捕获异常。那么PHP里的异常用法应该是什么样的呢？看下面的例子。

先定义两个异常类，它们需要继承自系统的exception，如代码清单1-18所示。

代码清单1-18　定义两个异常类

```
class emailException extends exception{
}
class pwdException extends exception{
function toString()
 {
return"<div class=\"error\">Exception{$this->getCode()}:
```

```
{$this->getMessage()}
in File: {$this->getFile()} on line: {$this->getLine()}</div>";
//改写抛出异常结果
}
}
```

然后就是实际的业务，根据业务需求抛出不同异常，如代码清单1-19所示。

代码清单1-19　根据业务需求抛出不同异常

```
function reg($reginfo=null) {
if(empty($reginfo)||!isset($reginfo)) {
throw new Exception("参数非法");
}
if(empty($reginfo['email'])) {
throw new emailException("邮件为空");
}
if($reginfo['pwd']!=$reginfo['repwd']) {
throw new pwdException("两次密码不一致");
}
echo'注册成功';
}
```

上面的代码判断传入的参数，根据业务进行异常分发。首先，如果没有传入任何参数（这个参数可以是POST进来，也可以是别的地方赋值得到），就把异常分发给exception超类，跳出注册流程；如果Email地址不存在，那么把异常分发给自定义的emailException异常，跳出注册流程；如果两次密码不一致，则将异常分发给自定义的pwdException，跳出注册流程。

现在异常分发了，但还不算完，还需要对异常进行分拣并做处理。代码如下所示：

```
try{
reg(array('email'=>'waitfox@qq.com','pwd'=>123456,'repwd'=>12345678));
//reg();
}catch(emailException$ee){
echo$ee->getMessage();
}catch(pwdException$ep){
echo$ep;
echo PHP_EOL,'特殊处理';
}catch(Exception$e){
echo$e->getTraceAsString();
echo PHP_EOL,'其他情况，统一处理';
}
```

这一段代码用于捕获所抛出的各种异常，进行分门别类的处理。

提示　可以尝试不同注册条件，看看异常分拣的流程。需要注意，exception作为超类应该放在最后捕获。不然，捕获这个异常超类后，后面的捕获就终止了，而这个超类不能提供针对性的信息和处理。

在这里，对表单进行异常处理，通过重写异常类、手动抛出错误的方式进行异常处理。这是一种业务异常，可以人为地把所有不符合要求的情况都视作业务异常，和通常意义上的代码异常相区别。

那PHP里的异常应该怎么用？在什么时候抛出异常，什么时候捕获呢？什么场景下能应用异常？在下面三种场景下会用到异常处理机制。

1.对程序的悲观预测

如果一个程序员对自己的代码有“悲观情绪”，这里并不是指该程序员代码质量不高，而是他认为自己的代码无法一一处理各种可预见、不可预见的情况，那该程序员就会进行异常处理。假设一个场景，程序员悲观地认为自己的这段代码在高并发条件下可能产生死锁，那么他就会悲观地抛出异常，然后在死锁时进行捕获，对异常进行细致的处理。

2.程序的需要和对业务的关注

如果程序员希望业务代码中不会充斥大堆的打印、调试等处理，通常他们会使用异常机制；或者业务上需要定义一些自己的异常，这个时候就需要自定义一个异常，对现实世界中各种各样的业务进行补充。比如上班迟到，这种情况认为是一个异常，要收集起来，到月底集中处理，扣你工资；如果程序员希望有预见性地处理可能发生的、会影响正常业务的代码，那么它需要异常。在这里，强调了异常是业务处理中必不可少的环节，不能对异常视而不见。异常机制认为，数据一致很重要，在数据一致性可能被破坏时，就需要异常机制进行事后补救。

举个例子，比如有个上传文件的业务需求，要把上传的文件保存在一个目录里，并在数据库里插入这个文件的记录，那么这两步就是互相关联、密不可分的一个集成的业务，缺一不可。文件保存失败，而插入记录成功就会导致无法下载文件；而文件保存成功数据库写入失败，则会导致没有记录的文件成为死文件，永远得不到下载。

那么假设文件保存成功后没有提示，但是保存失败会自动抛出异常，访问数据库也一样，插入成功没有提示，失败则自动抛出异常，就可以把这两个有可能抛出异常的代码段包在一个try语句里，然后用catch捕捉错误，在catch代码段里删除没有被记录到数据库的文件或者删除没有文件的记录，以保证业务数据的一致性。因此，从业务这个角度讲，异常偏重于保护业务数据一致性，并且强调对异常业务的处理。

如果代码中只是象征性地try……catch，然后打印一个报错，最后over。这样的异常不如不用，因为其没有体现异常思想。所以，合理的代码应该如下：

```
<? php
try {
//可能出错的代码段
if（文件上传不成功）throw（上传异常）;
if（插入数据库不成功）throw（数据库操作异常）;
} catch（异常） {
必须的补救措施，如删除文件、删除数据库插入记录，这个处理很细致
}
//……
? >
```

也可以如下：

```
<? php
上传 {
if（文件上传不成功）throw（上传异常）;
if（插入数据库不成功）throw（数据库操作异常）;
}
//其他代码……
try {
上传；其他；
} catch（上传异常） {
必须的补救措施，如删除文件，删除数据库插入记录
} catch（其他异常） {
记录log
}
? >
```

上面两种捕获异常的方式中，前一种是在异常发生时立刻捕获；后一种是分散抛异常集中捕获。那到底应该是哪一种呢？

如果业务很重要，那么异常越早处理越好，以保证程序在意外情况下能保持业务处理的一致性。比如一个操作有多个前提步骤，突然最后一个步骤异常了，那么其他前提操作都要消除掉才行，以保证数据的一致性。并且在这种核心业务下，有大量的代码来做善后工作，进行数据补救，这是一种比较悲观的、而又重要的异常。我们应把异常消灭在局部，避免异常的扩散。

如果异常不是那么重要，并且在单一入口、MVC风格的应用中，为了保持代码流程的统一，则常常采用后一种异常处理方式。这种异常处理方式更多强调业务流程的走向，对善后工作并不是很关心。这是一种次要异常，其将异常集中处理从而使流程更专一。

异常处理机制可以把每一件事当做事务考虑，还可以把异常看成一种内建的恢复系统。如果程序某部分失败，异常将恢复到某个已知稳定的点上，而这个点就是程序的上下文环境，而try块里面的代码就保存

catch所要知道的程序上下文信息。因此，如果很看重异常，就应该分散进行try……catch处理。

3.语言级别的健壮性要求

在健壮性这点上，PHP是不足的。以Java为例，Java是一种面向企业级开发的语言，强调健壮性。Java中支持多线程，Java认为，多线程被中断这种情况是彻彻底底的无法预料和避免的。所以Java规定，凡是用了多线程，就必须正视这种情况。你要么抛出，不管它；要么捕获，进行处理。总之，你必须面对InterruptedException异常，不准回避。也就是异常发生后应对重要数据业务进行补救，当然你可以不做，但是你必须意识到，异常有可能发生。

这类异常是强制的。更多异常是非强制的，由程序员决定。Java对异常的分类和约束，保证了Java程序的健壮性。

异常就是无法控制的运行时错误，会导致出错时中断正常逻辑运行，该异常代码后面的逻辑都不能继续运行。那么try……catch处理的好处就是：可以把异常造成的逻辑中断破坏降到最小范围内，并且经过补救处理后不影响业务逻辑的完整性；乱抛异常和只抛不捕获，或捕获而不补救，会导致数据混乱。这就是异常处理的一个重要作用，就是通过精确控制运行时的流程，在程序中断时，有预见地用try缩小可能出错的影响范围，及时捕获异常发生并做出相应补救，以使逻辑流程仍然能回到正常轨道上。

1.6.2 怎样看PHP的异常

PHP中的异常机制是不足的，绝大多数情况下无法自动抛出异常，必须用if……else先进行判断，再手动抛出异常。手动抛异常的意义不是很大，因为这意味着在代码里已经充分预期到错误的出现，也就算不上真正的“异常”，而是意料之中。同时，这种方式还会使你陷入纷繁复杂的业务逻辑判断和处理中。

Java语言做得比较好的就是定义了一堆内置的常见异常，不需要程序员判断各种异常情况后再手动抛出，编译器会代我们进行判断业务是否发生错误，若发生了，则自动抛出异常。作为程序员，只需要关心异常的捕获和随后补救，而不是像PHP那样关注到底会发生哪些异常，用if……else逐一判断，逐一抛出异常。

有没有什么机制使得PHP可以自动抛出异常呢？有，那就是结合PHP中的错误处理主动抛出异常。

使用异常能一定程度上会降低耦合性，但是也不能滥用。滥用异常的后果就是很可能导致代码被多处挂起，流程变得更复杂，难于理解。但是可以肯定，异常在PHP里有很大的价值，越复杂的应用，越需要合理考虑使用异常。

提示　需要提醒读者关注，SPL里定义了一大堆exception，如BadMethodCallException、LogicEx ception等，同时这些异常之间还存在层级关系。这些异常只是一个空壳，什么方法都没有，需要自己填充。它们实际上起到一个命名参考的作用。

1.6.3 PHP中的错误级别

错误处理本来不属于面向对象的范畴，但是既然讲到异常，就不得不提及异常的同胞兄弟——错误。

PHP错误处理比异常的价值大得多。PHP错误的概念已经和异常做过比较，这里通过对PHP异常的认知，给PHP错误下个最直观最通俗的结论：PHP错误就是会使脚本运行不正常的情况。

PHP错误有很多种，包括warning、notice、deprecated、fetal error等。这和一般意义的错误概念有些差别。所以，notice不叫通知，而叫通知级别的错误，warning也不叫警告，而叫警告级别的错误。

错误大致分为以下几类。

deprecated是最低级别的错误，表示“不推荐，不建议”。比如在PHP 5中使用ereg系列的正则匹配函数就会报此类错误。这种错误一般由于使用不推荐的、过时的函数或语法造成的。其虽不影响PHP正常流程，但一般情况下建议修正。

其次是notice。这种错误一般告诉你语法中存在不当的地方。如使用变量但是未定义就会报此错。最常见的，数组索引是字符时没有加引号，PHP就视为一个常量，先查找常量表，找不到再视为变量。虽然PHP是脚本语言，语法要求不严，但是仍然建议对变量进行初始化。这种错误不影响PHP正常流程。

warning是级别比较高的错误，在语法中出现很不恰当的情况时才会报此错误，比如函数参数不匹配。这种级别的错误会导致得不到预期结果，故需要修改代码。

更高级别的错误是fetal error。这是致命错误，直接导致PHP流程终结，后面的代码不再执行。这种问题非改不可。

最高级别的错误是语法解析错误prase error。上面提到的错误都属于PHP代码运行期间错误，而语法解析错误属于语法检查阶段错误，这将导致PHP代码无法通过语法检查。错误级别不止这几个，最主要的都在前面提到了。PHP手册中一共定义了16个级别的错误，最常用的就这

几个。代码清单1-20演示了常见级别的错误。

代码清单1-20　error.php

```
//Error.php
<? php
$date=' 2012-12-20' ;
if (ereg ("([0-9]{4})-([0-9]{1, 2})-([0-9]{1, 2})", $date, $regs)) {
echo"$regs[3].$regs[2].$regs[1]";
} else {
echo"Invalid date format: $date";
}
if ($i>5) {
echo' $i没有初始化啊' , PHP__EOL;
}
$a=array (' o' =>2, 4, 6, 8) ;
echo$a[o];
$result=array__sum ($a, 3) ;
echo fun () ;
echo' 致命错误后呢? 还会执行吗? ' ;
//echo' 最高级别的错误' , $55;
```

这段代码演示至少四个级别的错误，如果看不全，应确保你的php.ini文件做了如下设定：

```
error__reporting=E__ALL|E__STRICT
display__errors=On
```

error__reporting指定错误级别，上面的设置是最严格的错误级别，具体设置可以参php.ini。

提示　有一个技巧我想你会用到，那就是在代码质量或者环境不可控时（比如数据库连接失败），使用error__reporting（0），这样就能屏蔽错误了，正式部署时可以采取这样的策略，防止错误消息泄露敏感信息。另外一个技巧就是在函数前加@符号，抑制错误信息输出，如@mysql__connect（）。

1.6.4　PHP中的错误处理机制

PHP里有一套错误处理机制，可以使用set__error__handler接管PHP错误处理，也可以使用trigger__error函数主动抛出一个错误。

set__error__handler（）函数设置用户自定义的错误处理函数。函数用于创建运行期间的用户自己的错误处理方法。它需要先创建一个错误处理函数，然后设置错误级别。语法如下：

```
set__error__handler（error__function, error__types）
```

参数描述如下：

error__function：规定发生错误时运行的函数。必需。

error__types：规定在哪个错误报告级别会显示用户定义的错误。可选。默认为“E__ALL”。

提示　如果使用该函数，会完全绕过标准PHP错误处理函数，如果有必要，用户定义的错误处理程序必须终止（die（））脚本。

如果在脚本执行前发生错误，由于在那时自定义程序还没有注册，因此就不会用到这个自定义错误处理程序。这先实现一个自定义的异常处理函数，如代码清单1-21所示。

代码清单1-21　自定义的异常处理函数

```
<? php
function customError（$errno, $errstr, $errfile, $errline）
 {
echo"<b>错误代码：</b>[$ {errno} ]$ {errstr} \r\n";
echo"错误所在的代码行：｛$errline｝文件｛$errfile｝\r\n";
echo"PHP版本", PHP__VERSION, "（", PHP__OS, "）\r\n";
//die（）;
}
set__error__handler（"customError", E__ALL | E__STRICT）;
 $a=array（' o' =>2, 4, 6, 8）;
echo $a[o];
```

在这个函数里，可以对错误的详情进行格式化输出，也可以做任何要做的事情，比如判断当前环境和权限给出不同的错误提示，可使用errer__log函数将错误记入log文件，还可以细化处理，针对errno的不同

进行对应的处理。

自定义的错误处理函数一定要有这四个输入变量errno、errstr、errfile、errline。

errno是一组常量，代表错误的等级，同时也有一组整数和其对应，但一般使用其字符串值表示，这样语义更好一点。比如E__WARNING，其二进制掩码为4.，表示警告信息。

接下来，就是将这个函数作为回调参数传递给set__error__handler。这样就能接管PHP原生的错误处理函数了。要注意的是，这种托管方式并不能托管所有种类的错误，如E__ERROR、E__PARSE、E__CORE__ERROR、E__CORE__WARNING、E__COMPILE__ERROR、E__COMPILE__WARN ING，以及E__STRICT中的部分。这些错误会以最原始的方式显示，或者不显示。

set__error__handler函数会接管PHP内置的错误处理，你可以在同一个页面使用restore__error__handler（）；取消接管。

注意　如果使用自定义的set__error__handler接管PHP的错误处理，先前代码里的错误抑制@将失效，这种错误也会被显示。

在PHP异常中，异常处理机制是有限的，无法自动抛出异常，必须手动进行，并且内置异常有限。PHP把许多异常看做错误，这样就可以把这些“异常”像错误一样用set__error__handler接管，进而主动抛出异常。代码如下所示：

```
function customError（$errno，$errstr，$errfile，$errline）
 {
//自定义错误处理时，手动抛出异常
throw  new  Exception（$level.′ |′ .$ {errstr}）；
}
set__error__handler（"customError"，E__ALL | E__STRICT）；
try
 {
$a=5/0；
}
catch（Exception$e）
 {
echo′ 错误信息：′ ，$e->getMessage（）；
}
```

这样就能捕获到异常和非致命的错误，就能按照1.6.1节里讲述的方法进行处理了，这样可以弥补PHP异常处理机制的部分不足。

这种“曲折迂回”的处理方式存在的问题就是：必须依靠程序员自己

来掌控对异常的处理，对于异常高发区、敏感区，如果程序员处理不好，就会导致前面所提到的业务数据不一致的问题。其优点在于，可以获得程序运行时的上下文信息，以进行针对性的补救。

fetal error这样的错误虽然捕获不到，也无法在发生此错误后恢复流程处理，但是还是可以使用一些特殊方法对这种错误进行处理的。这需要用到一个函数——register__shutdown__function，此函数会在PHP程序终止或者die时触发一个函数，给PHP来一个短暂的“回光返照”。在PHP 4的时代，类不支持析构函数，常用这个函数模拟实现析构函数。实例代码如下：

```
<? php
class Shutdown
 {
public function stop ()
 {
if (error__get__last () )
 {
print__r (error__get__last () ) ;
}
die (' Stop.' ) ;
}
}
register__shutdown__function (array (new Shutdown () , ' stop' ) ) ;
$a=new a () ; //将因为致命错误而失败
echo' 必须终止' ;
```

可以运行看看效果。对于fetal error还能做点收尾工作，但是PHP流程的终止是必然的。对于Parse error级别的错误，你只有傻眼了，除了可以修改配置文件php.ini，什么都做不了，修改的内容如下：

```
log__errors=On
error__log=usr/log/php.log
```

这样一旦PHP发生了错误，就会被记入log文件，方便以后查询。

和exception类似，错误处理也有对应抛出错误的函数，那就是trigger__error函数，如下所示：

```
<? php
$divisor=0;
if ($divisor==0) {
trigger__error ("Cannot divide by zero", E__USER__ERROR) ;
}
echo' break' ;
```

关于错误处理，主要就是这些内容，还有一些错误处理和调试相关，我们将会放到后面的章节进行讲解。

提示　在PHP中，错误和异常是两个不同的概念，这种设计从根本上导致了PHP的异常和其他语言相异。以Java为例，Java中，异常是错误唯一的报告方式。说到底，两者的区别就是对异常和错误的认识不同而产生的。PHP的异常绝大部分必须通过某种办法手动抛出，才能被捕获到，是一种半自动化的异常处理机制。

无论是错误还是异常，都可以使用handler接管系统已有的处理机制。

1.7 本章小结

本章主要介绍面向对象思想的程序的组成元素——类和对象。类是一个动作和属性的模板，对象是数据的集合。结合PHP自身实际情况，着重讲述PHP里面向对象的一些比较模糊的知识点，包括魔术方法、接口、多态、类的复用、反射、异常机制等。接口是一种类型，从接口的实现讲述接口是怎么实现“即插即用”的。

然后，对异常机制进行探讨。讲述异常应该是什么样的，应该怎么用，并且阐述了PHP中的异常为什么会这样，应该在什么场合使用异常等。PHP起初没有异常机制，后期为了进军企业级开发，才模仿Java加进去的，故有了错误处理和异常处理的并存，这种形式导致PHP异常处理不伦不类，通过和Java对比，让我们了解到了异常的真实含义。错误处理是对异常处理的一种补充。

到底面向过程和面向对象孰优孰劣呢？答案是：二者间并无高低优劣之别，它们各有优劣。

其实在OO发展中，暴露出一些问题，如深入对象内部读写状态存在的困难，现实和开发中不对应造成的建模困难，数据与逻辑绑定造成的类型臃肿。比如前面提到的反射，就是因为面向对象的封装导致读写内部状态比较困难而产生的。

面向对象存在的问题是越来越多的语言引入函数式编程的特征，如闭包、回调等。PHP也引入一些函数式编程的概念，有兴趣的读者可以自行研究。

第2章　面向对象的设计原则

第1章　已经说过，面向对象是一种高度抽象的思维。在面向对象设计中，类是基本单位，各种设计都是围绕着类来进行的。可以说，类与类之间的关系，构成了设计模式的大部分内容。

在初学阶段，可以认为类就是属性+函数组成的，实际上在底层存储上也确实是这样的，但是，这些仅仅是确定一个独立的类。而类与类之间的关系是设计模式所要探讨的内容。

经典的设计模式有23种，每种都是对代码复用和设计的总结，就设计模式而言，除了熟读GOF经典外，推荐《敏捷软件开发——原则、方法与实践》一书。本章并不就具体的设计模式展开讨论，而是讨论一些基本的设计原则，并给出一些小的实例，最后，作为前两章的总结，探讨一下PHP中的面向对象的一些问题。

2.1　面向对象设计的五大原则

在面向对象的设计中，如何通过很小的设计改变就可以应对设计需求的变化，这是设计者极为关注的问题。为此不少OO先驱提出了很多有关面向对象的设计原则用于指导OO的设计和开发。下面是几条与类设计相关的设计原则。

面向对象设计的五大原则分别是单一职责原则、接口隔离原则、开放-封闭原则、替换原则、依赖倒置原则。这五大原则也是23种设计模式的基础[1]。

2.1.1　单一职责原则

亚当·斯密曾就制针业做过一个分工产生效率的例子[2]。对于一个没有受过相应训练，又不知道怎样使用这种职业机械的工人来讲，即使他竭尽全力地工作，也许一天连一根针也生产不出来，当然更生产不出20根针了。但是，如果把这个行业分成各种专门的组织，再把这种组织分成许多个部门，其中大部分部门也同样分为专门的组织。把制针分为18种不同工序，这18种不同操作由18个不同工人来担任。那么，尽管他们的机器设备都很差，但他们尽力工作，一天也能生产12磅针。每磅中等

型号针有4000根，按这个数字计算，十多个人每天就可以制造48000根针，而每个人每天能制造4800根针。如果他们各自独立地工作，谁也不专学做一种专门的业务，那么他们之中无论是谁都绝不可能一天制造20根针，也许连1根针也制造不出来。这就是企业管理中的分工，在面向对象的设计里，叫做单一职责原则（Single PesponsibilityPrinciple, SRP）.

在《敏捷软件开发》中，把“职责”定义为“变化的原因”，也就是说，就一个类而言，应该只有一个引起它变化的原因。这是一个最简单，最容易理解却最不容易做到的一个设计原则。说得简单一点，就是怎样设计类以及类的方法界定的问题。这种问题是很普遍的，比如在MVC的框架中，很多人会有这样的疑惑，对于表单插入数据库字段过滤与安全检查应该是放在control层处理还是model层处理，这类问题都可以归到单一职责的范围。

再比如在职员类里，将工程师、销售人员、销售经理等都放在职员类里考虑，其结果将会非常混乱。在这个假设下，职员类里的每个方法都要用if……else判断是哪种情况，从类结构上来说将会十分臃肿，并且上述三种职员类型，不论哪一种发生需求变化，都会改变职员类，这是我们所不愿意看到的！

从上面的描述中应该能看出，单一职责有两个含义：一个是避免相同的职责分散到不同的类中，另一个是避免一个类承担太多职责。

那为什么要遵守SRP呢？

（1）可以减少类之间的耦合

如果减少类之间的耦合，当需求变化时，只修改一个类，从而也就隔离了变化；如果一个类有多个不同职责，它们耦合在一起，当一个职责发生变化时，可能会影响其他职责。

（2）提高类的复用性

修理电脑比修理电视机简单多了。主要原因就在于电视机各个部件之间的耦合性太高，而电脑则不同，电脑的内存、硬盘、声卡、网卡、键盘灯部件都可以很容易地单独拆卸和组装。某个部件坏了，换上新的即可。

上面的例子就体现了单一职责的优势。由于使用了单一职责，使得“组件”可以方便地“拆卸”和“组装”。

不遵守SRP会影响对该类的复用性。当只需要复用该类的某一个职责时，由于它和其他的职责耦合在一起，也就很难分离出。

遵守SRP在实际代码开发中有没有什么应用？有的。以数据持久层为例，所谓的数据持久层主要指的是数据库操作，当然，还包括缓存管理等。以数据库操作为例，如果是一个复杂的系统，那么就可能涉及多种数据库的相互读写等，这时就需要数据持久层支持多种数据库。应该怎么做？定义多个数据库操作类？你的想法已经很接近了，再进一步，就是使用工厂模式。

工厂模式（Factory）允许你在代码执行时实例化对象。它之所以被称为工厂模式是因为它负责“生产”对象。以数据库为例，工厂需要的就是根据不同的参数，生成不同的实例化对象。最简单的工厂就是根据传入的类型名实例化对象，如传入MySQL，就调用MySQL的类并实例化，如果是SQLite，则调用SQLite的类并实例化，甚至可以处理TXT、Excel等“类数据库”。工厂类也就是这样的一个类，它只负责生产对象，而不负责对象的具体内容。

先定义一个接口，规定一些通用的方法，如代码清单2-1所示。

代码清单2-1　定义一个适配器接口

```
<? php
interface Db__Adapter {
/**
*数据库连接
*@param  $config数据库配置
*@return resource
*/
public function connect（$config）;
/**
*执行数据库查询
*@param string$query数据库查询SQL字符串
*@param mixed$handle连接对象
*@return resource*/
public function query（$query, $handle）;
}？>
```

这是一个简化的接口，并没有提供所有方法，其定义了MySQL数据库的操作类，这个类实现了Db__Adapter接口，具体如代码清单2-2所示。

代码清单2-2　定义MySQL数据库的操作类

```
<? php
class Db__Adapter__Mysql implements Db__Adapter
 {
private $__dbLink; //数据库连接字符串标示
/**
*数据库连接函数
*
*@param $config数据库配置
*@throws Db__Exception
*@return resource*/
public function connect ( $config)
 {
if ( $this->__dbLink=@mysql__connect ( $config->host.
 (empty ( $config->port) ? ' ' : ' : ' .$config->port),
 $config->user, $config->password, true) ) {
if (@mysql__select__db ( $config->database, $this->__dbLink) ) {
if ( $config->charset) {
mysql__query ("SET NAMES' {$config->charset}' ", $this->__dbLink);
}
return $this->__dbLink;
} }
/**数据库异常*/
throw new Db__Exception (@mysql__error ( $this->__dbLink) );
}
/**
*执行数据库查询
*
*@param string $query数据库查询SQL字符串
*@param mixed $handle连接对象
*@return resource
*/
public function query ( $query, $handle)
 {
if ( $resource=@mysql__query ( $query, $handle) ) {
return $resource;
}
}
} ? >
```

接下来是SQLite数据库的操作类，同样实现了Db__Adapter接口，如代码清单2-3所示。

代码清单2-3　SQLite数据库的操作类

```
<? php
class Db__Adapter__sqlite implements Db__Adapter
 {
private $__dbLink; //数据库连接字符串标示
/**
*数据库连接函数
*@param $config数据库配置
*@throws Db__Exception
*@return resource*/
public function connect ( $config)
 {
if ( $this->__dblink=sqlite__open ( $config->file, 0666, $error) ) {
return $this->__dblink;
}
/**数据库异常*/
throw new Db__Exception ( $error);
}
/**
*执行数据库查询
*@param string $query数据库查询SQL字符串
*@param mixed $handle连接对象
*@return resource
*/
public function query ( $query, $handle)
 {
if ( $resource=@sqlite__query ( $query, $handle) ) {
return $resource;
}
}
}
```

好了，如果现在需要一个数据库操作的方法的话怎么做？只需定义

一个工厂类，根据传入不同的参数生成需要的类即可，如代码清单2-4所示。

代码清单2-4　定义一个工厂类

```
<? php
class sqlFactory
 {
public static function factory（$type）
 {
if（include_once′ Drivers/′ .$type.′ .php′ ） {
 $classname=′ Db_Adapter_′ .$type；
return new$classname；
} else {
throw new Exception（′ Driver not found′ ）；
}
}
} ? >
```

要调用时，就可以这么写：

```
$db=sqlFactory：factory（′ MySQL′ ）；
$db=sqlFactory：factory（′ SQLite′ ）；
```

我们把创建数据库连接这块程序单独拿出来，程序中的CURD就不用关心是什么数据库了，只要按照规范使用对应的方法即可。

工厂方法让具体的对象解脱了出来，使其并不再依赖具体的类，而是抽象。除了数据库操作这种显而易见的设计外，还有什么地方会用到工厂类呢？那就是SNS中的动态实现。

下面的图片来自国内某SNS网站，属于当前新鲜事页面，可以看到针对不同行为，其生成了不同动态。比如，参加了某个小组，动态显示的就是“XX参加了YY小组”；收到某某的礼物，别人看到的多台就是“XX收到了YY的ZZ礼物”，如图2-1所示。

以上这种动态应该怎么设计呢，最容易想到的就是用工厂模式，根据传入的操作不同，结合模板而生成不同的动态，如代码清单2-5所示。

图　2-1　某SNS网站的动态展示

代码清单2-5　工厂模式

```
<bead id="feedServiceFactory"class="FeedServiceFactory">
<property name="feedMap">
<map>
<entry key="friend"value-ref="friendFeed"/>
<entry key="album"value-ref="albumFeed"/>
<entry key="reply"value-ref="replyFeed"/>
<entry key="share"value-ref="shareFeed"/>
<entry key="video"value-ref="videoFeed"/>
<entry key="group"value-ref="groupFeed"/>
</map>
</property>
</bean>
```

以上代码是一个动态的生成配置，通过FEED的类型匹配到key，取到对应的bean，然后创建不同的动态，用的就是工厂模式。

设计模式里面的命令模式也是SRP的体现，命令模式分离“命令的请求者”和“命令的实现者”方面的职责。举一个很好理解的例子，就是你去餐馆吃饭，餐馆存在顾客、服务员、厨师三个角色。作为顾客，你只要列出菜单，传给服务员，由服务员通知厨师去实现。作为服务员，只需要调用准备饭菜这个方法（对厨师大喊“该炒菜了”），厨师听到要炒菜的请求，就立即去做饭。在这里，命令的请求和实现就完成了解耦。

模拟这个过程，首先定义厨师角色，厨师进行实际的做饭、烧汤的工作。详细代码如代码清单2-6所示。

代码清单2-6　餐馆的示例

```
/**
厨师类，命令接受者与执行者
***/
class cook {
public function meal() {
echo' 番茄炒鸡蛋' , PHP_EOL;
}
public function drink() {
echo' 紫菜蛋花汤' , PHP_EOL;
}
public function ok() {
echo' 完毕' , PHP_EOL;
} }
//然后是命令接口
interface Command {
//命令接口
public function execute();
}
```

现在轮到服务员出场，服务员是命令的传送者，通常你到饭馆吃饭都是叫服务员吧，不可能直接叫厨师，一般都是叫“服务员，给我来盘番茄炒西红柿”，而不会直接叫“厨师，给我来盘番茄炒西红柿”。所以，服务员是顾客和厨师之间的命令沟通者。模拟这个过程的代码如代码清单2-7所示。

代码清单2-7　模拟服务员与厨师的过程

```
class MealCommand implements Command {
private$cook;
//绑定命令接受者
public function construct(cook$cook) {
$this->cook=$cook;
}
public function execute() {
$this->cook->meal(); //把消息传递给厨师，让厨师做饭，下同
}
}
class DrinkCommand implements Command {
private$cook;
//绑定命令接受者
public function construct(cook$cook) {
$this->cook=$cook;
}
public function execute() {
$this->cook->drink();
} }
```

现在顾客可以按照菜单叫服务员了，如代码清单2-8所示。

代码清单2-8　模拟顾客与服务员的过程

```
class cookControl {
private$mealcommand;
private$drinkcommand;
//将命令发送者绑定到命令接收器上面来
public function addCommand(Command$mealcommand, Command$drinkcommand) {
$this->mealcommand=$mealcommand;
$this->drinkcommand=$drinkcommand;
}
public function callmeal() {
$this->mealcommand->execute();
}
public function calldrink() {
```

```
$this->drinkcommand->execute();
}}
```

好了，现在完成整个过程，如代码清单2-9所示。

代码清单2-9　实现命令模式

```
$control=new cookControl;
$cook=new cook;
$mealcommand=new MealCommand($cook);
$drinkcommand=new DrinkCommand($cook);
$control->addCommand($mealcommand, $drinkcommand);
$control->callmeal();
$control->calldrink();
```

从上面的例子可以看出，原来设计模式并非纯理论的东西，而是来源于实际生活，就连普通的餐馆老板都懂设计模式这门看似高深的学问。其实，在经济和管理活动中，对流程的优化就是对各种设计模式的摸索和实践。所以，设计模式并非计算机编程中的专利。事实上，设计模式的起源不是计算机学科，而是源于建筑学。

在设计模式方面，不仅以上这两种体现了SRP，还有别的（比如代理模式）也体现了SRP。SRP不只是对类设计有意义，对以模块、子系统为单位的系统架构设计同样有意义。

模块、子系统也应该仅有一个引起它变化的原因，如MVC所倡导的各个层之间的相互分离其实就是SRP在系统总体设计中的应用。图2-2是来自CI框架的流程图。

SRP是最简单的原则之一，也是最难做好的原则之一。我们会很自然地将职责连接在一起。找到并且分离这些职责是软件设计需要达到的目的。

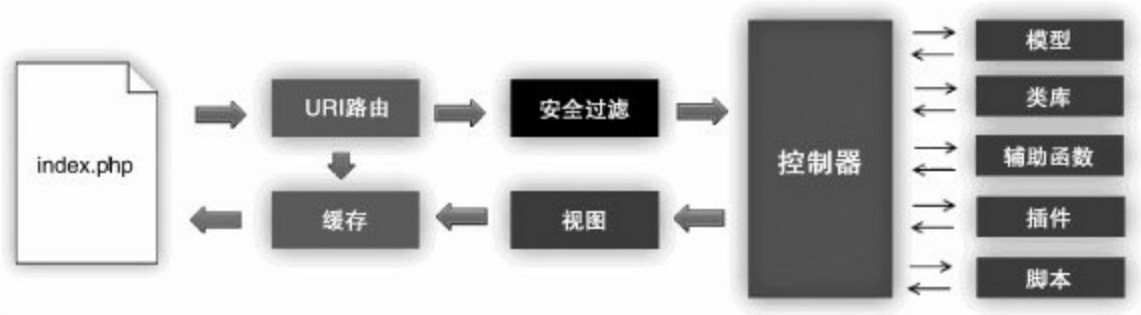

图　2-2　MVC中的流程

一些简单的应该遵循的做法如下：

根据业务流程，把业务对象提炼出来。如果业务流层的链路太复杂，就把这个业务对象分离为多个单一业务对象。当业务链标准化后，对业务对象的内部情况做进一步处理。把第一次标准化视为最高层抽象，第二次视为次高层抽象，以此类推，直到“恰如其分”的设计层次。

职责的分类需要注意。有业务职责，还要有脱离业务的抽象职责，从认识业务到抽象算法是一个层层递进的过程。就好比命令模式中的顾客，服务员和厨师的职责，作为老板（即设计师）的你需要规划好各自的职责范围，既要防止越俎代庖，也要防止互相推诿。

[1]这些原则主要是由Robert C.Martin在《敏捷软件开发——原则、方法与实践》一书中总结出来的。
[2]见《国富论》第1章，分工理论是亚当·斯密的一个重要经济理论。

2.1.2 接口隔离原则

设计应用程序的时候，如果一个模块包含多个子模块，那么我们应该小心对该模块做出抽象。设想该模块由一个类实现，我们可以把系统抽象成一个接口。但是要添加一个新的模块扩展程序时，如果要添加的模块只包含原系统中的一些子模块，那么系统就会强迫我们实现接口中的所有方法，并且还要编写一些哑方法。这样的接口被称为胖接口或者被污染的接口，使用这样的接口将会给系统引入一些不当的行为，这些不当的行为可能导致不正确的结果，也可能导致资源浪费。

1.接口隔离

接口隔离原则（Interface Segregation Principle, ISP）表明客户端不应该被强迫实现一些他们不会使用的接口，应该把胖接口中的方法分组，然后用多个接口代替它，每个接口服务于一个子模块。简单地说，就是使用多个专门的接口比使用单个接口要好得多。

ISP的主要观点如下：

1）一个类对另外一个类的依赖性应当是建立在最小的接口上的。

ISP可以达到不强迫客户（接口的使用方）依赖于他们不用的方法，接口的实现类应该只呈现为单一职责的角色（遵守SRP原则）。

ISP还可以降低客户之间的相互影响——当某个客户程序要求提供新的职责（需求变化）而迫使接口发生改变时，影响到其他客户程序的可能性会最小。

2）客户端程序不应该依赖它不需要的接口方法（功能）。

客户端程序不应该依赖它不需要的接口方法（功能），那依赖什么？依赖它所需要的接口。客户端需要什么接口就提供什么接口，把不需要的接口剔除，这就要求对接口进行细化，保证其纯洁性。

比如在应用继承时，由于子类将继承父类中的所有可用的方法；而父类中的某些方法，在子类中可能并不需要。例如，普通员工和经理都继承自雇员这个接口，员工需要每天写工作日志，而经理则不需要。因

此不能用工作日志来卡经理，也就是经理不应该依赖于提交工作日志这个方法。

可以看出，ISP和SRP在概念上是有一定交叉的。事实上，很多设计模式在概念上都有交叉，甚至你很难判断一段代码属于哪一种设计模式。

ISP强调的是接口对客户端的承诺越少越好，并且要做到专一。当某个客户程序的要求发生变化，而迫使接口发生改变时，影响到其他客户程序的可能性小。这实际上就是接口污染的问题。

2.对接口的污染

过于臃肿的接口设计是对接口的污染。所谓接口污染就是为接口添加不必要的职责，如果开发人员在接口中增加一个新功能的主要目的只是减少接口实现类的数目，则此设计将导致接口被不断地“污染”并“变胖”。

接口污染会给系统带来维护困难和重用性差等方面的问题。为了能够重用被污染的接口，接口的实现类就被迫要实现并维护不必要的功能方法。

“接口隔离”其实就是定制化服务设计的原则。使用接口的多重继承实现对不同的接口的组合，从而对外提供组合功能——达到“按需提供服务”。

看下面这个例子，如图2-3所示。

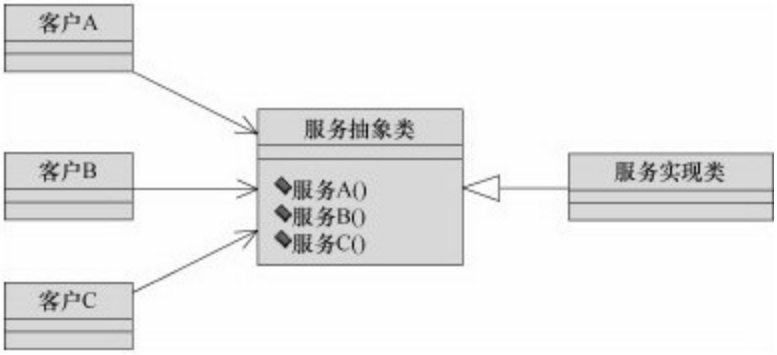

图　2-3　存在污染的接口设计

客户A需要A服务，只要针对客户A的方法发生改变，客户B和客户C就会受到影响。故这种设计需要对接口进行隔离，如图2-4所示。

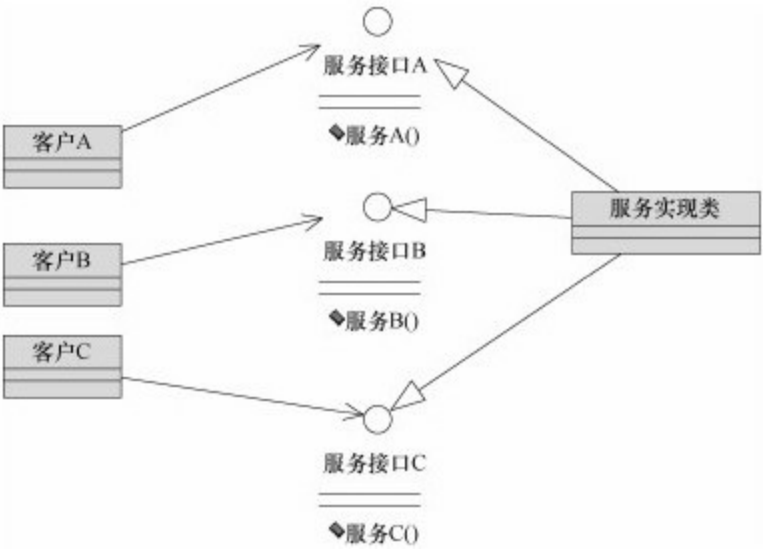

图　2-4　减少接口中的污染

由图2-4可知，如果针对客户A的方法发生改变，客户B和客户C并不会受到任何影响。你可能会想，这样做接口那岂不是会很多？这个问题问得很好，接口既要拆，但也不能拆得太细，这就得有个标准，这就是高内聚。接口应该具备一些基本的功能，能独一完成一个基本的任务。

图2-4所示只是个抽象的例子，在实际应用中，会遇到如下问题：比如，我需要一个能适配多种类型数据库的DAO实现，那么首先应实现一个数据库操作的接口，其中规定一些数据库操作的基本方法，如连接数据库、增删查改、关闭数据库等。这是一个最少功能的接口。对于一些MySQL中特有的而其他数据库不具有或性质不同的方法，如PHP里可能用到的MySQL的pconnect方法，其他数据库里并不存在和这个方法相同的概念，这个方法也就不应该出现在这个基本的接口里，那这个基本的接口应该有哪些基本的方法呢？PDO已经告诉你了。

PDO是一个抽象的数据接口层，它告诉我们一个基本的数据库操作接口应该实现哪些基本的方法。接口是一个高层次的抽象，所以接口里的方法应该是通用的、基本的、不易变化的。

还有一个问题，那些特有的方法应该怎么实现？根据ISP原则，这些方法可以在另一个接口中存在，让这个“异类”同时实现这两个接口。

对于接口的污染，可以考虑下面这两条处理方式：

利用委托分离接口。

利用多继承分离接口。

委托模式中，有两个对象参与处理同一个请求，接受请求的对象将请求委托给另一个对象来处理，如策略模式、代理模式等中都应用到了委托的概念。至于其实现，在反射那一节其实已经实现了，这里就不再细讲了。

利用多继承分离接口，在接口一节也做了相应的讲解，这里不再重复。

2.1.3 开放-封闭原则

1.什么是“开放-封闭”

随着软件系统的规模不断增大，软件系统的维护和修改的复杂性不断提高，这种困境促使法国工程院院士Bertrand Meyer在1998年提出了“开放-封闭”（Open Close Principle, OCP）原则，这条原则的基本思想是：

Open（Open for extension）模块的行为必须是开放的、支持扩展的，而不是僵化的。

Closed（Closed for modification）在对模块的功能进行扩展时，不应该影响或大规模地影响已有的程序模块。

换句话说，也就是要求开发人员在不修改系统中现有功能代码（源代码或者二进制代码）的前提下，实现对应用系统的软件功能的扩展。用一句话概括就是：一个模块在扩展性方面应该是开放的而在更改性方面应该是封闭的。

从生活中，最容易想到的例子就是电脑，我们可以轻松地对电脑进行功能的扩展，而只需通过接口连入不同的设备。

开放-封闭能够提高系统的可扩展性和可维护性，但这也是相对的，对于一台电脑不可能完全开放，有些设备和功能必须保持稳定才能减少维护上的困难。要实现一项新的功能，你就必须升级硬件，或者换一台更高性能的电脑。以电脑中的多媒体播放软件为例，作为一款播放器，应该具有一些基本的、通用的功能，如打开多媒体文件，停止播放、快进、音量调节等功能。但不论是什么播放器，不论是在什么平台下，遵循这个原则设计的播放器都应具有统一风格和操作习惯，无论换用哪一款播放器，都应保证操作者能快速上手。

以播放器为例，先定义一个抽象的接口，代码如下所示。

```
interface process {
public function process () ;
}
```

然后，对此接口进行扩展，实现解码和输出的功能，如代码清单2-10所示。

代码清单2-10　实现播放器的编码功能

```
class playerencode implements process{
public function process() {
echo"encode\r\n";
}
}
class playeroutput implements process{
public function process() {
echo"output\r\n";
}
}
```

对于播放器的各种功能，这里是开放的，只要你遵照约定，实现了process接口，就能给播放器添加新的功能模块。这里只实现解码和输出模块，还可以依据需求，加入更多新的模块。

接下来为定义播放器的线程调度管理器，播放器一旦接收到通知（可以是外部单击行为，也可以是内部的notify行为），将回调实际的线程处理，如代码清单2-11所示。

代码清单2-11　播放器的“调度管理器”

```
class playProcess{
private$message=null;
public function construct() {
}
public function callback(event$event) {
$this->message=$event->click();
if($this->message instanceof process) {
$this->message->process();
}}
}
```

具体的产品出来了，在这里定义一个MP4类，这个类是相对封闭的，其中定义事件的处理逻辑，如代码清单2-12所示。

代码清单2-12　播放器的事件处理逻辑

```
class mp4{
public function work() {
$playProcess=new playProcess();
$playProcess->callback(new event('encode'));
$playProcess->callback(new event('output'));
}
}
```

最后为事件分拣的处理类，此类负责对事件进行分拣，判断用户或

内部行为，以产生正确的“线程”，供播放器内置的线程管理器调度，如代码清单2-13所示。

代码清单2-13　播放器的事件处理类

```
class event {
private$m;
public function construct（$me） {
$this->m=$me;
}
public function click（） {
switch（$this->m） {
case′ encode′ ：
return new playerencode（）；
break;
case′ output′ ：
return new playeroutput（）；
break;
}
}
}
```

最后，运行下下面的代码：

```
$mp4=new mp4;
$mp4->work（）；
```

输出结果如下：

```
encode output
```

这就实现了一个基本的播放器，此播放器的功能模块是对外开放的，但是内部处理应该是相对封闭和稳定的。但这个实现还存在一些问题，这就需要你来发现了。有时候为了降低系统的复杂性，也会不完全遵守设计模式，而是对其进行增删改。

2.如何遵守开放-封闭原则

实现开放-封闭的核心思想就是对抽象编程，而不对具体编程，因为抽象相对稳定。让类依赖于固定的抽象，这样的修改就是封闭的；而通过面向对象的继承和对多态机制，可以实现对抽象体的继承，通过覆写其方法来改变固有行为，实现新的扩展方法，所以对于扩展就是开放的。

1）在设计方面充分应用“抽象”和“封装”的思想。

一方面也就是要在软件系统中找出各种可能的“可变因素”，并将之封装起来；

另一方面，一种可变性因素不应当散落在多个不同代码模块中，而应当被封装到一个对象中。

2）在系统功能编程实现方面应用面向接口的编程。

当需求发生变化时，可以提供该接口新的实现类，以求适应变化。

面向接口编程要求功能类实现接口，对象声明为接口类型。在设计模式中，装饰模式比较明显地用到了OCP。

2.1.4 替换原则

替换原则由MIT计算机科学实验室的Liskov女士在1987年的OOPSLA大会上的一篇文章《Data Abstraction and Hierarchy》中提出，主要阐述有关继承的一些原则，故又称里氏替换原则。

2002年，Robert C.Martin出版了一本名为《Agile Software Development Principles Patterns and Practices》的书，在书中他把里氏代换原则最终简化为一句话：“Subtypes must be substitutable for their base types”。（子类必须能够替换成它们的基类。）

1.LSP的内容

里氏替换原则（Liskov Substitution Principle, LSP）的定义和主要的思想如下：由于面向对象编程技术中的继承在具体的编程中过于简单，在许多系统的设计和编程实现中，我们并没有认真地、理性地思考应用系统中各个类之间的继承关系是否合适，派生类是否能正确地对其基类中的某些方法进行重写等问题。因此经常出现滥用继承或者错误地进行了继承等现象，给系统的后期维护带来不少麻烦。这就需要我们有一个设计原则来遵循，它就是替换原则。

LSP指出：子类型必须能够替换掉它们的父类型、并出现在父类能够出现的任何地方。它指导我们如何正确地进行继承与派生，并合理地重用代码。此原则认为，一个软件实体如果使用一个基类的话，那么一定适用于其子类，而且这根本不能察觉出基类对象和子类对象的区别。想一想，是不是和第一章提到的多态的概念比较像？

2.LSP主要是针对继承的设计原则

因为继承与派生是OOP的一个主要特性，能够减少代码的重复编程实现，从而实现系统中的代码复用，但如何正确地进行继承设计和合理地应用继承机制呢？

这就是LSP所要解决的问题：

如何正确地进行继承方面的设计？

最佳的继承层次如何获得？

怎样避免所设计的类层次陷入不符合OCP原则的状况？

那如何遵守该设计原则呢？

父类的方法都要在子类中实现或者重写，并且派生类只实现其抽象类中声明的方法，而不应当给出多余的方法定义或实现。

在客户端程序中只应该使用父类对象而不应当直接使用子类对象，这样可以实现运行期绑定（动态多态）。

如果A、B两个类违反了LSP的设计，通常的做法是创建一个新的抽象类C，作为两个具体类的超类，将A和B的共同行为移动到C中，从而解决A和B行为不完全一致的问题。

在前面的多态，继承这几节的内容里，已经涉及LSP，包括使用多态实现隐藏基类和派生类对象的区别，以及使用组合的方式解决继承中的基类与派生类（即子类）中的不符合语意的情况。PHP对LSP的支持并不好，缺乏向上转型等概念，只能通过一些曲折的方法实现。对于这个原则，这里就不再细讲了。

在接口那节提到了一个缓存的实现接口，试试用抽象类做基类，遵循LSP实现其设计。

这里给出其抽象类代码，如代码清单2-14所示。

代码清单2-14　缓存实现抽象类

```
<? php
abstract class Cache {
/**
*设置一个缓存变量
*
*@param String $key 缓存Key
*@param mixed $value 缓存内容
*@param int $expire缓存时间（秒）
*@return boolean是否缓存成功
*/
public abstract function set（$key, $value, $expire=60）;
/**
*获取一个已经缓存的变量
*@param String $key缓存Key
*@return mixed缓存内容
*/
public abstract function get（$key）;
/**
*删除一个已经缓存的变量
*@return boolean  是否删除成功
*/
public abstract function del（$key）;
/**
```

```
*删除全部缓存变量
*
*@return boolean 是否删除成功
*/
public abstract function delAll();
/**
*检测是否存在对应的缓存
**/
public abstract function has($key);
}
```

如果现在要求实现文件、memcache、accelerator等各种机制下的缓存，只需要继承这个抽象类并实现其抽象方法即可。

现在，再来思考本书开头提到的白马非马的问题，试着用里氏替换原则阐释。

注意　LSP中代换的不仅仅是功能，还包括语意。试思考：白马可以代换马，而牛同样作为劳力，可代换马否？高跟鞋也是鞋子，男人穿高跟鞋又是否能接受？

2.1.5 依赖倒置原则

什么是依赖倒置呢？简单地讲就是将依赖关系倒置为依赖接口，具体概念如下：上层模块不应该依赖于下层模块，它们共同依赖于一个抽象（父类不能依赖子类，它们都要依赖抽象类）。

抽象不能依赖于具体，具体应该要依赖于抽象。

注意，这里的接口不是狭义的接口。

为什么要依赖接口？因为接口体现对问题的抽象，同时由于抽象一般是相对稳定的或者是相对变化不频繁的，而具体是易变的。因此，依赖抽象是实现代码扩展和运行期内绑定（多态）的基础：只要实现了该抽象类的子类，都可以被类的使用者使用。这里，我想强调一下扩展性这个概念。通常扩展性是指对已知行为的扩展，在讲述接口那一节，我也提到，接口应该是相对稳定的。这就告诉我们，无论使用多么先进的设计模式，也无法做到不需要修改代码即可达到以不变应万变的地步。在面向对象的这五大原则里，我认为依赖倒置是最难理解，也是最难实现的。

这个例子以前面提到的雇员类为蓝本，实现代码如代码清单2-15所示。

代码清单2-15　employee.php

```
<? php
interface employee {
public function working () ;
}
class teacher implements employee {
public function working () {
echo' teaching……' ;
}
}
class coder implements employee {
public function working () {
echo' coding……' ;
}
}
class workA {
public function work () {
$teacher=new teacher;
$teacher->working () ;
}
}
class workB {
private$e;
public function set (employee$e) {
$this->e=$e;
}
public function work () {
$this->e->working () ;
}
}
```

```
$worka=new workA;
$worka->work();
$workb=new workB;
$workb->set(new teacher());
$workb->work();
```

在classA中，work方法依赖于teacher实现；在classB中，work转而依赖于抽象，这样可以把需要的对象通过参数传入。上述代码通过接口，实现了一定程度的解耦，但仍然是有限的。不仅是使用接口，使用工厂等也能实现一定程度的解耦和依赖倒置。

在workB中，teacher实例通过setter方法传入中，从而实现了工厂模式。由于这样的实现仍然是硬编码的，为了实现代码的进一步扩展，把这个依赖关系写在配置文件里，指明classB需要一个teacher对象，专门由一个程序检测配置是否正确（如所依赖的类文件是否存在）以及加载配置中所依赖的实现，这个检测程序，就称为IOC容器。

很多文章里看到IOC（Inversion of Control）概念，实际上，IOC是依赖倒置原则（Depend ence Inversion Principle, DIP）的同义词。而在提IOC的时候，你可能还会看到有人提起DI等概念。DI，即依赖注入，一般认为，依赖注入（DI）和依赖查找（DS）是IOC的两种实现。不过随着某些概念的演化，这几个概念之间的关系也变得很模糊，也有人认为IOC就是DI。有人认为，依赖注入的描述比起IOC来更贴切，这里不纠缠于这几个概念之间的关系。

在经典的J2EE设计里，通常把DAO层和Service层细分为接口层和实现层，然后在配置文件里进行依赖关系的配置，这是最常见的DIP的应用。Spring框架就是一个很好的IOC容器，把控制权从代码剥离到IOC容器，这里是通过XML配置文件实现的，Spring在执行时期根据配置文件的设定，建立对象之间的依赖关系。

如下面代码所示：

```
<bean scope="prototype"
class="cn.notebook.action.NotebookListOtherAction"id="notebookListOtherAction">
<property ref="userReplyService"name="userReplyService"/><property ref="userService"name="userService"/>
<property ref="permissionService"name="permissionService"/><property ref="friendService"name="friendService"/>
</bean>
```

但是这样设置一样存在问题，配置文件会变得越来越大，其间关系会越来越复杂。同样逃脱不了随着应用和业务的改变，不断修改代码的恶魇（这里认为配置文件是代码的一部分。并且在实际开发中，很少存

在单纯修改配置文件的情况。一般配置文件修改了，代码也会做相应修改）。

在PHP里，也有类似模仿Spring的实现，即把依赖关系写在了配置文件里，通过配置文件来产生需要的对象。我觉得这样的代码是还是为了实现而实现。在Spring里，配置文件里配置的不仅仅是一个类运行时的依赖关系，还可以实现事务管理、AOP、延迟加载等。而PHP要实现上面的种种特性，其消耗是巨大的。从语言层面讲，PHP这种动态脚本型语言在实现一些多态特性上和编译型的语言不同。其次PHP作为敏捷性的开发语言，更强调快速开发、逻辑清晰、代码简单易懂，如果再附加了各种设计模式的框架，从技术实现和运行效率上来看，都是不可取的。依赖倒置的核心原则是解耦。如果脱离这个最原始的原则，那就是本末倒置。

事实上，很多的设计模式里已经隐含了依赖倒置原则，我们也在有意或无意地做着一些依赖反转的工作。只是作为PHP，目前还没有一个比较完善的IOC容器，或许是PHP根本不需要。

如何满足DIP：

每个较高层次类都为它所需要的服务提出一个接口声明，较低层次类实现这个接口。

每个高层类都通过该抽象接口使用服务。

2.2　一个面向对象留言本的实例

在这一节，用面向对象的思想完成一个简单的留言本模型，这个模型不涉及实际的数据库操作以及界面显示，只是一个demo，用来演示面向对象的一些思维。

在面向过程的思维里，要设计一个留言本，一切都将以留言本为核心，抓到什么是什么，按流程走下来，即按用户填写信息→留言→展示的流程进行。

现在用面向对象的思维思考这个问题，在面向对象的世界，会想尽办法把肉眼能看见的以及看不见的，但是实际存在的物或者流程抽象出来。既然是留言本，那么就存在留言内容这个实体，这个留言实体（domain）应该包括留言者的姓名、E-mail、留言内容等要素，如代码清单2-16所示。

代码清单2-16　留言实体类message.php

```
class message{
public$name;//留言者姓名
public$email;//留言者联系方式
public$content;//留言内容
public function set($name,$value){
$this->$name=$value;
}
public function get($name){
if(!isset($this->$name)){
$this->$name=NULL;
}
}
}
```

上面的类也就是所说的domain，是一个真实存在的、经过抽象的实体模型。然后，需要一个留言本模型，这个留言本模型包括留言本的基本属性和基本操作，如代码清单2-17所示。

代码清单2-17　留言本模型gbookModel.php

```
/**
*留言本模型，负责管理留言本
*$bookPath:留言本属性
*/
class gbookModel{
private$bookPath;//留言本文件
private$data;//留言数据
public function  setBookPath($bookPath){
$this->bookPath=$bookPath;
}
public  function getBookPath(){
return$this->bookPath;
}
```

```
public function open（） {
}
public function  close（） {
}
public function read（） {
return file_get_contents（$this->bookPath）;
}
//写入留言
public function write（$data） {
$this->data=self：safe（$data）->name."&".self：safe（$data）->email."\r\nsaid：\r\n".self：safe
（$data）->content;
return
file_put_contents（$this->bookPath，$this->data, FILE_APPEND）;
}
//模拟数据的安全处理，先拆包再打包
public static function safe（$data） {
$reflect=new ReflectionObject（$data）;
$props  =$reflect->getProperties（）; $messagebox=new stdClass（）;
foreach（$props as$prop） {
$ivar=$prop->getName（）;
$messagebox->$ivar=trim（$prop->getValue（$data））;
}
return$messagebox;
}
public function delete（） {
file_put_contents（$this->bookPath，' it' s empty now' ）; }
}
```

实际留言的过程可能会更复杂，可能还包括一系列准备操作以及Log处理，所以应定义一个类负责数据的逻辑处理，如代码清单2-18所示。

代码清单2-18　留言本业务逻辑处理leaveModel.php

```
class leaveModel{
public function write（gbookModel$gb，$data） {
$book=$gb->getBookPath（）;
$gb->write（$data）;
//记录日志
}
}
```

最后，通过一个控制器，负责对各种操作的封装，这个控制器是直接面向用户的，所以包括留言本查看、删除、留言等功能。可以形象理解为这个控制器就是留言本所提供的直接面向使用者的功能，封装了操作细节，只需要调用控制器的相应方法即可，如代码清单2-19所示。

代码清单2-19　前端控制部分代码

```
class authorControl{
public function message（leaveModel$l, gbookModel$g, message$data） {
//在留言本上留言
$l->write（$g，$data）;
}
public function view（gbookModel$g） {
//查看留言本内容
return$g->read（）;
}
public function delete（gbookModel$g） {
$g->delete（）;
echo self：view（$g）;
}
}
```

测试代码如下所示：

```
$message=new message;
$message->name='phper';
$message->email='phper@php.net';
$message->content='a crazy phper love php so much.';
$gb=new authorControl();//新建一个留言相关的控制器
$pen=new leaveModel();//拿出笔
$book=new gbookModel();//翻出笔记本
$book->setBookPath("g:\\bak\\temp\\tempcode\\a.txt");
$gb->message($pen, $book, $message);
echo$gb->view($book);
$gb->delete($book);
```

这样看起来是不是比面向过程要复杂多了？确实是复杂了，代码量增多了，也难以理解了。似乎也体现不出优点来。但是你思考过以下问题吗？

如果让很多人来负责完善这个留言本，一部分负责实体关系的建立，一部人负责数据操作层的代码，这样是不是更容易分工了呢？

如果我要把这个留言本进一步开发，实现记录在数据库中，或者添加分页功能，又该如何呢？

要实现上面第二个问题提出的功能，只需在gbookModel类中添加分页方法，代码如下所示：

```
public function readByPage() {
$handle=file($this->bookPath);$count=count($handle);
$page=isset($__GET['page'])?intval($__GET['page']):1;
if($page<1||$page>$count)$page=1;
$pnum=9;
$begin=($page-1)*$pnum;
$end=($begin+$pnum)>$count?$count:$begin+$pnum;
for($i=$begin;$i<$end;$i++) {
echo'<strong>',$i+1,'</strong>',$handle[$i],'<br/>';
}
for($i=1;$i<=ceil($count/$pnum);$i++) {
echo"<a href=?page=${i}>${i}</a>";
}}
```

然后到前端控制器里添加对应的action，代码如下所示：

```
public function viewByPage(gbookModel$g) {
return$g->readByPage();
}
```

运行结果如图2-5所示。

只需要这么简单的两步，就可实现所需要的分页功能，而且已有的方法都不用修改，只需在相关类中新增方法即可。当然，这个分页在实

际点击时是有问题的，因为我没有把Action分开，而是通通放在一个页面里。对照着上面的思路，还可以把留言本扩展为MySQL数据库的。

在这个程序里只体现了非常简单的设计模式，这个程序还有许多要改进的地方，每个程序员心中都有一个自己的OO。项目越大越能体现模块划分、面向对象的好处。

图　2-5　程序运行结果

思考　试着找找这个小程序里体现了哪些设计原则，并且试着加上一些异常处理等。

2.3 面向对象的思考

PHP的特色是简单、快速、适用。在PHP的世界里，一切以解决问题为主，所以很多设计方面的东西往往被忽视或排斥。虽然PHP的面向对象提出很多年了，但一直被排斥，很多人提倡原生态开发方式，甚至有人提倡彻底面向过程。伴随着对OO的质疑，PHP框架一方面如雨后春笋般遍地开花，另一方面一直受到抵制和质疑。

有一点是肯定的，PHP不是一门很好的面向对象的语言，因为其无法做到完全面向对象，也无法优雅实现面向对象。所以现在比较流行的还是以类为主的开放方式，即抛弃或精简经典的MVC理论，很少用和几乎不用设计模式，以类加代码模块的方式进行代码组织。这种开发方式在PHP的开源项目里是最流行的，也是最适合二次开发的，而比较纯的面向对象的产品有Zend Framework。这类产品入门的门槛比较高，代码看似“臃肿”，开发成本比较高，这类产品一般比较少见，市场占有率也比较低。

所有产品最终都是为市场服务的，PHP面向的是Web开发市场，所以并不需要高端的、复杂的设计和开发技巧。但是前面讲的那些并不是没有作用。

一些基本理论，在任何一门语言里都有共性。语法和函数库只是学好一门语言的必要条件，而不是充要条件。语法和函式只是表层的东西。只要掌握面向对象的思想，即使没有一点Java和.NET基础，也能看懂用它们写成的代码。

PHP只是一个脚本语言、一门工具而已。在Web开发中，PHP语言自身所占的分量越来越低，但却涉及程序设计的方方面面，而面向对象只是其中之一，也是最主要的一个方面。PHP是一种经典思想，能实现低耦合、易扩展的代码，其可用最经济的方式干一件事。

理论是重要的，但是理论也不是一成不变的。比如我们提到的一些设计模式，也没必要完全遵守，可以做一些精简和变形。

基于以上思考，我们认为在PHP的开发中应该灵活使用面向对象的特性和设计原则。

对于流程明确、需求清晰、需求变更风险小的业务逻辑，过程化开发（传统软件开发模式）最适合，这就像解一道数学题，总需要一步步去解，上一步的结果作为下一步的条件。这个时候，面向过程的开发更符合人的思维。

但是对于流程复杂、需求不完善、存在很大需求变更风险的业务逻辑，此时用过程化开发将使程序变得非常的繁琐臃肿，实现难度很大，并且后期的维护代价高得惊人。此时，抽象思维将是最适合的，用面向对象的思维去抽象业务模型并随需求不断精化，最终交付使用，其扩展度和可维护性都要比过程化方法更好。

由于面向对象是更高一层的抽象，它有一些优点较之面向过程是比较突出的：

其一，新成员的加入和融合不再困难，高度抽象有利于高度总结。

其二，代码即文档，团队中的任何人都可以轻松地获得产品各个模块的基本信息，而不再需要通读大部分代码。

说到这里，可能就会有人有疑问了：本书一直在推崇面向对象的开发模式，说面向对象的好，说OO适合复杂的项目，那Linux这种复杂的项目，使用面向过程的C语言编写的，这又如何解释？

这个问题问得好，现解释如下：

其一，Linux虽然是用面向过程的C语言编写的，但是Linux的操作系统是使用内核+模块的方式构建的，这种模块化的思想是所有编程范式中的普适原则。

其二，面向对象和各种设计模式就是已经提供好的模式，使用已有的模式本比像Linux那样自己摸索出一个模式更方便快捷，开发成本更低，代码更易阅读。

其实，面向过程也好，面向对象也好，目的只有两个：一个是功能实现，一个是代码维护和扩展。只要能做好这两点，那就是成功的。

PHP不是一门很好的OOPL，但却是一门很好的Web设计语言。我们有理由相信，在Web开发领域，PHP还将继续发挥其作用，以其简

单、快速吸引更多的开发者加入。

2.4 本章小结

本章主要讲解面向对象设计的五大原则，穿插一些设计模式的例子。在第1章的最后提到面向对象的设计思想存在一些问题，其本质在于面向对象强调对现实的建模，而现实和开发中并没有一一对应，因此五大原则和设计模式就是对OO的补充。

最后一节给出的留言本demo，只是一个很小的模型。一般来说，越是规模较大的项目，越能体现设计模式的前瞻性和必要性。

可能很多读者对一些设计模式有不同的见解和困惑，这是正常的。一段代码往往很难明确地归属于某一种设计模式，其可能有多种设计模式的影子。设计模式只是一种成熟的、可供借鉴的思考模式，而不是公式。

我们既要深入了解面向对象的思想，又不能执着于面向对象。

第3章　正则表达式基础与应用

正则表达式起源于科学家对人类神经系统工作原理的早期研究。美国新泽西州的Warren McCulloch和出生在美国底特律的Walter Pitts这两位神经生理方面的科学家，研究出一种用数学方式来描述神经网络的新方法，他们创新地将神经系统中的神经元描述成小而简单的自动控制元，从而做出一项伟大的工作革新。后来，数学科学家Stephen Kleene在Warren McCulloch和Walter Pitts早期工作的基础之上发表一篇论文，题目是《神经网事件的表示法》，书中利用正则集合的数学符号描述此模型，引入正则表达式的概念。

3.1　认识正则表达式

正则表达式就是用某种模式去匹配一类字符串的一种公式。通俗地讲，就是用一个“字符串”描述一个特征，然后验证另一个“字符串”是否符合这个特征的公式。

比如“ab+”描述的特征是：一个a和任意个b。那么ab、abb、abbbbbbbbbb都符合这个特征，而字符串ad显然是不符合的。

正则表达式可应用到各个方面。在常用的高级编辑器中，几乎都支持正则表达式，如Word、EditPlus、UltraEdit、Vim等。

正则表达式在编程语言中更是得到大规模推广。现在的语言几乎都是原生的，都可从语法上支持正则表达式，尤其在Perl的推动下，PHP、Java、.NET、JavaScript等语言都支持丰富的正则语法；不支持的可以通过一些包实现扩展。每种语言中对正则表达式的支持有所不同，其中Perl和.NET对正则表达式的支持最为强大，而JavaScript对正则表达式的支持则比较“朴素”。

注意　本节所讲的一些特性，并不是在所有语言中都支持。

3.1.1　PHP中的正则函数

正则表达式看起来总是那么古怪，以至于许多人对其望而生畏。首先要澄清一些概念：虽然不同语言间正则语法大同小异，但实际上正则

表达式的实现有多种引擎（如非确定性有穷自动机NFA、确定性有穷自动机DFA），其表现又有多种风格（如JavaScript有自己的朴素正则、Perl有一套高级而强大的正则、.NET也有自己的一套正则风格）。另外，还有人可能容易混淆PHP中的preg和ereg。

简单地说，PHP中有两套正则函数，两者功能差不多：

1）由PCRE库提供的函数，以“preg__”为前缀命名。

PCRE（Perl Compatible Regular Expression，兼容Perl的正则表达式）由Philip Hazel于1997年开发。现代的编程语言和软件中一般都使用PCRE库。

2）由POSIX扩展提供的函数，以“ereg__”为前缀命名。

POSIX（Portable Operating System Interface of UNIX, UNIX可移植操作系统接口）由一系列规范构成，定义了UNIX操作系统应支持的功能，所以“POSIX风格的正则表达式”也就是“关于正则表达式的POSIX规范”，定义了BRE（Basic Regular Expression，基本型正则表达式）和ERE（Extended Regular Express，扩展型正则表达式）两大流派。通常UNIX的一些工具和较老的软件中会使用POSIX风格的正则。另外，一些数据库中也提供了POSIX风格的正则表达式。

自PHP 5.3以后，就不再推荐使用POSIX正则函数库，若程序中使用了则会报Deprecated级别的错误，这种情况通常在一些较老的代码中比较常见。其实使用或不使用POSIX正则函数库二者本质上没多大差别，主要是一些表现形式、语法和扩展功能的差别。

3.1.2 正则表达式的组成

在Windows资源管理器中查找文件以及批处理文件时，可使用通配符“？”和“*”表示匹配一组字符，这和正则表达式类似，“？”表示一个不确定的字符，而“*”则表示任意多个不确定字符。比如下面是删除本地垃圾文件批处理的部分代码：

```
del/f/s/q%systemdrive%\*.tmp
del/f/s/q%systemdrive%\*.__mp
del/f/s/q%systemdrive%\*.log
del/f/s/q%systemdrive%\*.gid
del/f/s/q%systemdrive%\*.chk
del/f/s/q%systemdrive%\*.old
```

需要注意的是，这里的“？”和“*”称为“通配符”，而不是正则表达式。

在PHP里，一个正则表达式分为三个部分：分隔符、表达式和修饰符。

分隔符：可以是除了字母、数字、反斜线及空白字符以外的任何字符（比如/、！、#、%、｜、～等）。经常使用的分隔符是正斜线（/）、hash符号（#）以及取反符号（～）。考虑到可读性，为了避免和反斜线混淆，一般不使用正斜线做分隔符。

表达式：由一些特殊字符和非特殊的字符串组成，比如“[a-z0-9__ --]+@[a-z0-9__ .]+”可以匹配一个简单的电子邮件字符串。

修饰符：用于开启或者关闭某种功能/模式。

3.1.3　测试工具的使用

在学习过程中，建议下载RegexTester工具验证和测试正则表达式，也可使用Firefox的扩展Regular Expression Tester进行测试，其界面如图3-1所示。

图　3-1　Firefox的扩展Regular Expression Tester

本书以后测试都将利用此工具进行，而不再写PHP代码测试。

注意　这个工具测试的代码不一定能在PHP中通过，反之PHP中合法的正则表达式在此工具里也不一定能测试通过。其中的道理前面已经讲过了，不同语言实现的正则表达式略有区别。

下面，就来开始最简单的正则表达式入门的介绍。

3.2 正则表达式中的元字符

假设要在一篇文章里查找“he”，可以使用正则表达式“he”。这几乎是最简单的正则表达式，它可以精确匹配这样的字符串：由两个字符组成，前一个字符是“h”，后一个是“e”。通常，处理正则表达式的工具会提供一个忽略大小写的选项，如果选中这个选项，它可以匹配“he”、“HE”、“He”、“hE”这四种情况中的任意一种。

但是很多单词里包含“he”这两个连续的字符，比如“her”、“heet”等。用“he”来查找，这些单词中的“he”也会被找出来。如果要精确地查找“he”这个单词，应该使用以下形式：

```
\bhe\b
```

“\b”是正则表达式规定的一个特殊代码，代表单词的开头或结尾，也就是单词的分界处。虽然通常英文单词是由空格、标点符号或者换行来分隔，但是“\b”并不匹配这些单词分隔字符中的任何一个，它只匹配一个位置。

“\b”匹配位置的精确说法：前一个字符和后一个字符不全是（一个是，一个不是或不存在）“\w”。

假如要找“he”后面不远处跟着一个“is”，应该表示如下：

```
\bhe\b.*\bis\b
```

这里，点号（.）是元字符，匹配除了换行符以外的任意字符。“*”同样是元字符，不过它代表的不是字符，也不是位置，而是数量——它指定“*”前边的内容可以连续重复使用任意次以使整个表达式得到匹配。因此，“.”和“*”连在一起就意味着任意数量的、不包含换行的字符。现在，“\bhe\b.*\bis\b”的意思很明显：先是一个单词he，然后是任意个任意字符（但不能是换行符），最后是is这个单词。

3.2.1 什么是元字符

元字符（Meta　Characters）是正则表达式中具有特殊意义的专用字符，用来规定其前导字符（即位于元字符前面的字符）在目标对象中的出现模式。通过前面的例子，我们已经知道几个很有用的元字符。正则表达式里有很多元字符，常用元字符如表3-1所示。

表3-1　常用元字符

元字符	描　述
.	匹配除换行符以外的任意字符
\w	匹配字母或数字或下划线或汉字
\s	匹配任意空白符
\d	匹配数字
\b	匹配单词的开始或结束
^	匹配字符串的开始
$	匹配字符串的结束
-	表示范围
[]	匹配括号中的任意一个字符
*、+、?	量词

下面看一些例子。

1）匹配以字母“a”开头的单词：

```
\ba\w*\b
```

以上表达式先是某个单词开始处（\b），然后是字母“a”，接着是任意数量的字母或数字（\w*），最后是单词结束处（\b），匹配的单词如adandon、action、a等。

2）匹配1个或更多连续的数字：

```
\d+
```

以上表达式可以匹配0、1、555等。这里的元字符+和*类似，不同的是，*匹配重复任意次（可能是0次），而+则匹配重复1次或更多次。

3）匹配刚好6个字符的单词：

```
\b\w{6}\b
```

以上表达式匹配action、123456、ste__ph等。

注意　正则表达式里“单词”指不少于1个的连续字母和数字。

如果同时使用其他元字符，则能构造出功能更强大的正则表达式。比如下面这个例子：

```
0\d\d-\d\d\d\d\d\d\d\d
```

匹配字符串：以0开头，然后是2个数字，1个连字符，最后是8个数字，也就是中国部分地区的电话号码，如010 12345678。

这里“＼d”是元字符，匹配1位数字（0、1、2……）。“-”不是元字符，只匹配它本身——连字符（或者减号，或者中横线，或者随你怎么称呼它）。

为了避免那么多烦人的重复，也可以这样写这个表达式：

```
0\d{2}-\d{8}
```

这里＼d后面｛2｝和｛8｝的意思是，前面＼d必须连续重复匹配2次和8次。

思考题　使用“he”、“＼bhe＼b”分别查找句子“he is a good student, the most proud of his moth er.With him, she hold the hope.”有多少种匹配结果？

下面重点介绍几个常用元字符。

3.2.2 起始和结束元字符

元字符中有两个用来匹配位置：

^：匹配字符串的开始。

：匹配字符串的结束。

元字符“^”、“”与“\b”有点类似。“^”匹配字符串的开头，“”匹配结尾。这两个代码在验证输入内容时非常有用，比如某网站如果要求填写QQ号必须为5～11位数字时，可以使用：

```
^\d{5,11}$
```

这里{5，11}表示重复次数不能少于5次，不能多于11次，否则都不匹配。因为使用“^”和“”，所以输入的整个字符串都要和\d{5，11}匹配。也就是说，整个输入必须是5～11个数字，如果输入QQ号能匹配这个正则表达式，就符合要求。如果输入含有5～11个数字，但不是完整数字串，而只是一串字符的一部分，也不能匹配成功，如图3-2所示。

图 3-2 正则表达式匹配结果

从图中就能清晰地看出“^\d{5，11}”的确切含义。我想，你也能猜测到它和正则表达式\d{5，11}的区别。为了加深印象，分别使

用下面4个正则表达式看一下效果：

```
^\d{5，11}$//匹配起始和结束位置都是数字的，且连续5～11位
\d{5，11}$//匹配结束位置是数字的，且连续5～11位
^\d{5，11}//匹配起始位置是数字的，且连续5～11位
\d{5，11}//匹配连续的5～11位数字
```

很自然，在一行中，前三个正则表达式结果只可能有一个匹配结果，而最后一个正则表达式则可以有多个匹配成功的结果。因为一行只可能有一个开始位置和一个结束位置。

注意　我们在正则表达式处理工具处勾选Multiline选项，即多行选项，^和的意义就变成匹配行的开始处和结束处，否则将把整个输入视作一个字符串，忽视换行符。可以试着把多行选项去除后再看看效果。如果用过Vim编辑器，就知道命令“d^”和“d”的作用了。

3.2.3 点号

点号（.）是使用频率最高的元字符。例如，在做采集时抓取页面，要匹配某DIV里的内容，就需要用到点号匹配。下面代码是抓取本地HTML页面的一部分：

```
<html xmlns="http：//www.w3.org/1999/xhtml">
<head profile="http：//gmpg.org/xfn/11">
<meta http-equiv="content-type"content="text/html；charset=UTF-8"/>
<title>我的博客</title>
```

要匹配这个网页的标题应该怎么办呢？很简单，使用点号匹配全部字符，如下：

```
<title>.*<\/title>
```

这样就可以抓取你想要的任何内容了，包括DIV、SPAN等。

思考题　延伸思路，是不是还可以抓取页面的字符集？要判断这个页面有多少张图片是不是也很容易？只要找到特征字符就可以。试一下，看看和预想的结果是否一致。

3.2.4 量词

前面实际上已经涉及量词的概念，比如\d+、\d{5，11}等都应用了量词。正则表达式中的量词如表3-2所示。

下面是一些例子：

1）匹配Windows后面跟1个或更多数字：

```
Windows\d+
```

2）表示index后面紧跟0个或1个数字，：

```
index\d?
```

以上表达式匹配index、index1、index9这样的文件名，但不匹配index10、indexa这样的文件名。

3）匹配一行第一个单词（或整个字符串第一个单词，具体匹配哪种，得看选项设置）：

表3-2 正则表达式中的量词

限定符代码/语法	描　述
*	重复0次或更多次
+	重复1次或更多次
?	重复0次或1次
{n}	重复n次
{n,}	重复n次或更多次
{n,m}	重复n到m次

```
^\w+
```

提示　在学习量词的过程中，要注意*和？这两个量词。前面提到过通配符的概念，通配符里也有这两个符号，要注意它们之间的区别。

3.3 正则表达式匹配规则

我们已经学习“*”、“-”、“？”等元字符，它们都有各自的特殊含义。如果想匹配没有预定义元字符的字符集合，或者表达式和已知定义相反，或者存在多种匹配情况，应该怎么办？本节就介绍几种常用匹配规则。

3.3.1 字符组

查找数字、字母、空白很简单，因为已经有了对应这些字符集合的元字符，但是如果想匹配没有预定义元字符的字符集合（比如元音字母a、e、i、o、u），方法很简单，只需要在方括号里列出它们。

例如[aeiou]匹配任何一个英文元音字母，[.？！]匹配标点符号（“.”、“？”或“！”），c[aou]t匹配“cat”、“cot”、“cut”这三个单词，而“caout”则不匹配。

注意[]匹配单个字符，尽管看起来[]里有好多字符。

也可以指定字符范围，例如[0-9]的含意与＼d完全一致：代表一位数字；同理[a-z0-9A-Z__]完全等同于＼w（如果只考虑英文）。

字符组很简单，但是一定要弄清楚字符组中什么时候需要转义。

3.3.2 转义

如果想查找或匹配元字符本身，比如查找*、？等就出现问题：没办法指定，因为它们会被解释成别的意思。这时就使用\来取消这些字符的特殊意义。因此，应该使用\.和*。当然，查找\本身用\\。这叫做转义。

通俗地讲，转义就是防止特殊字符被解析，或者说用某个符号表示另一个特殊符号。例如：unibetter\.com匹配unibetter.com, C:\\Windows匹配C:\Windows。

在JavaScript或者PHP中都接触过转义的概念。例如，JavaScript中要弹出一个对话框，对话框中需要分成两行显示，用HTML的
标签或者在源代码里手工换行都不行，应该用\r\n表示换行并新起一行，如下所示：

```
alert("警告:
操作无效");//错误
alert("警告<br>操作无效");//错误
alert("警告\r\n操作无效");//正确写法
```

在PHP里使用反斜杠（\）表示转义，\Q和\E也可以在模式中忽略正则表达式元字符，比如：

```
\d+\Q.$.\E$
```

以上表达式先匹配一个或多个数字，紧接着一个点号，然后一个，再然后一个点号，最后是字符串末尾。也就是说，\Q和\E之间的元字符都会作为普通字符用来匹配。

正则表达式是不是遇到这些特殊字符就该转义呢？答案显然是否定的。转义只有在一定条件下，比如可能引起歧义或者被误解析的情况下才需要。有些情况并不需要转义这些“特殊”字符，并且在时转义也是无效的。这需要不断尝试并积累经验。看一个例子：

```
<?php
$reg="#[aby\]]#";
$str='a\bc[]{}';
preg_match_all($reg,$str,$m);
var_dump($m);
```

在字符组中匹配“a”、“b”、“y”和“｝”中任意一个，由于“｝”是元字符，具有特殊意义，所以这里进行转义，使用“＼｛”表示“｛”。

但是实际上，这个转义是多余的。虽然“｝”是元字符，具有特殊意义，但是在字符组中，“｝”却无法发挥意义，不会引起歧义，所以不需要转义。在这里“＼｛”和“｛”是等价的。

既然转义符“＼”是多余的，那么会不会被当作普通字符呢？字符串str里有“＼”，但是可以从代码运行结果中看出，“＼”字符并没有被匹配，也就是说正则表达式“#[abc＼｝]#”中，虽然“＼”转义符是多余的，但是也并没有被当作普通字符进行匹配。

如果确实要把“＼”当作普通字符匹配，正则表达式需要写成：

```
#[｝ab＼＼＼y]#
```

前面提到，不是所有出现特殊字符的地方都要转义。例如，以下正则表达式可以匹配“cat”、“c？t”、“c）t”等字符：

```
c[aou？*）]t
```

其中“？”和“*”等特殊字符都不需要转义。原因很简单，字符组里匹配的是单个字符，这些特殊字符不会引起歧义。

字符组里可以使用转义吗？可以，例如“c[＼d]d”可以匹配“c1d”、“c2d”等。下面是复杂的表达式：

```
＼（？0＼d｛2｝[）-]？＼d｛8｝
```

“（”和“）”也是元字符（后面在分组章节会提到），所以在这里需要使用转义。这个表达式可以匹配几种格式的电话号码，例如（010）88886666、022 22334455或02912345678等。首先是转义符“＼（”，表示出现0或1次（？），然后是一个0，后面跟着两个数字（＼d｛2｝），然后是“）”、“-”或空格中的一个，出现1次或不出现（？），最后是八个数字（＼d｛8｝）。

3.3.3 反义

有些时候，查找的字符不属于某个字符类，或者表达式和已知定义相反（比如除了数字以外其他任意字符），这时需要用到反义。常用反义如表3-3所示。

表 3-3 常用反义

常用反义	描 述
\W	匹配任意不是字母、数字、下画线、汉字的字符
\S	匹配任意不是空白符的字符
\D	匹配任意非数字的字符
\B	匹配不是单词开头或结束的位置
[^x]	匹配除了 x 以外的任意字符
[^aeiou]	匹配除了 aeiou 这几个字母以外的任意字符

反义有一个比较明显的特征，就是和一些已知元字符相反，并且为大写形式。比如“\d”表示数字，而“\D”就表示非数字。看一些实际的例子。

1）不包含空白符的字符串：

```
\S+
```

2）用尖括号括起来、以a开头的字符串：

```
<a[^>]+>
```

比如，要匹配字符串“百度”，这个正则表达式匹配的结果就是“<a-href="http：//baidu.com">”。

提示“^”在这里是“非”的意思，不要和表示开头的“^”混淆。那怎么区分呢？很简单，表示开始位置的“^”只能用在正则表达式最前端，而表示取反的“^”只用在字符组中，即只在中括号里出现。记住这一点，就不会搞混了。

日常工作中反义用得不多，因为扩大了范围。例如程序里的变量，第一个字符不允许是数字，一般使用“^[a-zA-Z__]”表示，而不会使

用“＼D”，因为“＼D”扩大了范围，包括所有非数字的字符，显然，变量命名不仅仅要求第一个字符不是数字，也不能是其他除了26个大小写字母和下画线以外的字符。因此，不要随意使用反义，以免无形中扩大范围，而使自己没有考虑到。

3.3.4 分支

分支就是存在多种可能的匹配情况。例如，匹配“cat”或者“hat”，可以写成[ch]at；要匹配“cat”、“hat”、“fat”、“toat”，很显然不能用字符组匹配的方式。这里表明前面的匹配字符可以是c、h、f或者to，而[]只能匹配单个字符，此时可用分支形式，即：

```
(c|h|f|to)at
```

其中括号里的表达式将视作一个整体（后面会讲到分组的概念），“|”表示分支，即可能存在的多种情况，可以匹配多个字符。分支的功能更强大，字符组方式只能对单个字符“分支”，而分支可以是多个字符以及更复杂的表达式。但对于单字符的情况，字符组的效率更高。也就是说，能使用字符组就不用分支。

看到这里，你可能会有疑问：表达式“[ch]at”括号里面是可能的匹配，分支也是表示可能的匹配，那么“[ch]at”是否可以写成“（c|h）at”呢？答案显然是可以的，“[ch]at=（c|h）at”。

注意　括号匹配会捕获文本，如果不需要捕获文本，上面的例子可以使用“（？：）”，后面还会讲到。

正则表达式分支条件指有几种规则，无论满足其中哪一种规则都能匹配，具体方法是用“|”把不同规则分隔开，例如：

```
0\d{2}-\d{8}|0\d{3}-\d{7}
```

这个表达式能匹配两种以连字号分隔的电话号码：一种是3位区号，8位本地号（如010-12345678），一种是4位区号，7位本地号（如0376-2233445）。匹配3位区号的电话号码表达式如下：

```
\(0\d{2}\)[-]?\d{8}|0\d{2}[-]?\d{8}
```

其中区号可以用小括号括起来，也可以不用，区号与本地号间可以用连字号或空格间隔，也可以没有间隔。可以试试用分支条件把这个表

达式扩展成同时支持4位区号。

例如，美国邮编规则是5位数字，或者用连字号间隔的9位数字。匹配表达式如下：

```
\d{5}-\d{4}|\d{5}
```

另外，使用分支条件时，要注意各个条件的顺序。如果改成以下形式，就只匹配5位邮编以及9位邮编的前5位：

```
\d{5}|\d{5}-\d{4}
```

注意　匹配分支条件时，将从左到右测试每个条件，如果满足某个分支，就不会再考虑其他条件。

3.3.5 分组

重复单个字符只需要直接在字符后面加上限定符，但如果想重复多个字符又该怎么办呢？可以用小括号指定子表达式，然后规定这个子表达式的重复次数，也可以对子表达式进行其他一些操作。这就是本节介绍的分组，常用分组语法如表3-4所示。

表 3-4 常用分组语法

类别	代码/语法	描 述
捕获	(exp)	匹配 exp，并捕获文本到自动命名的组里
	(?<name>exp)	匹配 exp，并捕获文本到名称为 name 的组里，也可以写成 (?'name'exp)
	(?:exp)	匹配 exp，不捕获匹配的文本，也不给此分组分配组号
零宽断言	(?=exp)	匹配 exp 前面的位置
	(?<=exp)	匹配 exp 后面的位置
	(?!exp)	匹配后面跟的不是 exp 的位置
	(?<!exp)	匹配前面不是 exp 的位置
注释	(?#comment)	提供注释辅助阅读，不对正则表达式的处理产生任何影响

例如，简单的IP地址匹配表达式如下：

```
(\d{1,3}\.){3}\d{1,3}
```

要理解以上表达式，应按下列顺序分析：

1）匹配1～3位的数字：

```
\d{1,3}
```

2）匹配3位数字加上1个英文句号（分组），重复3次（最后加上一个1～3位的数字）：

```
(\d{1,3}\.){3}
```

IP地址中每个数字都不能大于255，所以严格来说这个正则表达式是有问题的。因为它将匹配256.300.888.999这种不可能存在的IP地址。如果能使用算术比较，或许能简单地解决这个问题，但是正则表达式中

没有提供关于数学的任何功能，所以只能使用冗长的分组、选择、字符类来描述一个正确IP地址，如下所示：

```
（（2[0-4]\d|25[0-5]|[01]?\d\d?）\.）{3}（2[0-4]\d|25[0-5]|[01]?\d\d?）
```

思考题　理解这个表达式的关键是理解“2[0-4]＼d｜25[0-5]｜[01]?＼d＼d?”，读者应该能分析出它的意义。

默认情况下，每个分组会自动拥有一个组号，规则是：从左向右，以分组的左括号为标志，第一个出现的分组，其组号为1，第二个为2，以此类推；分组0对应整个正则表达式。

也可以自己指定子表达式的组名，语法如下：

```
?＜Word＞\w+
```

把尖括号换成单引号也行，如下所示：

```
?′Word′\w+
```

这样就把＼w+组名指定为Word。

提示　组号分配远没有这么简单。组号分配过程是要从左向右扫描两遍：第一遍只给未命名组分配，第二遍只给命名组分配。因此，所有命名组的组号都大于未命名的组号。可以使用语法（？：exp）剥夺一个分组对组号分配的参与权。

3.3.6 反向引用

反向引用用于重复搜索前面某个分组匹配的文本。首先看示例，“\1”代表分组1匹配的文本：

```
\b(\w+)\b\s+\1\b
```

以上表达式可以匹配重复的单词，例如go go或者kitty kitty。首先这个表达式是一个单词，也就是单词开始处和结束处之间大于一个的字母或数字，即“\b（\w+）\b”，这个单词会被捕获到编号为1的分组中，然后是1个或几个空白符（\s+），最后是分组1中捕获的内容（也就是前面匹配的那个单词），即\1，这样就相当于把所匹配的重复一次。

要反向引用分组捕获的内容，可以使用“\k<Word>”，所以上个例子也可以写成这样：

```
\b(?<Word>\w+)\b\s+\k<Word>\b
```

例如，要捕获字符串“\"This is a′ string′ \""引号内的字符，如果使用以下正则表达式：

```
(\"|′).*?(\"|′)
```

将返回“"This is a′ ”。显然，这并不是我们想要的内容。这个表达式从第一个双引号开始匹配，遇到单引号之后就错误地结束匹配。这是因为表达式里包含"|′，也就是双引号（"）和单引号（′）均可。要修正这个问题，可以用到反向引用。

表达式“\1，\2，……，\9”是对前面已捕获子内容的编号，可以作为对这些编组的“指针”引用。在此例中，第一个匹配的引号就由1代表。可以这么写成：

```
("|\′).*?\1
```

如果使用命名捕获组，可以写成：

```
（？P<quote>"|'）.*？（？P=quote）
```

看PHP使用反向引用的例子。

在很多论坛中都会看到UBB标签代码。UBB标签最早的设计是用来在论坛和留言本里代替HTML，实现一些简单的HTML效果，同时防止滥用HTML出现安全问题。例如，HTML中粗体的标签是：

```
<b>粗体</b>
```

或者：

```
<strong>粗体</strong>
```

而UBB标签则是：

```
[b]粗体[/b]
```

UBB标签以其更好的安全性，目前已经成为论坛发帖的代码标准，只不过不同论坛产品的叫法不一样而已。

最终，UBB标签还是要解析成HTML代码，才能让浏览器认识。这个过程是怎样实现的呢？下面以URL标签为例解释。

例如，UBB标签“[url]1.gif[/url]”用于插入表情。在解析时，需要把1.gif换成实际路径，并且需要用HTML的IMG标签进行替换，方法如下所示：

```
<？php
$str='[url]1.gif[/url][url]2.gif[/url][url]3.gif[/url]';
$s=preg_replace（"#\[url\]（？<WORD>\d\.gif）\[\/url\]#"，"<img src=http：//image.ai.com/upload/$1
>"，$str）；
var_dump（$s）；
```

运行结果如下：

```
string（141）"<img src=http：//image.ai.om/upload/1.gif>
<img src=http：//image.ai.com/upload/2.gif>
<img src=http：//image.ai.com/upload/3.gif>"
```

是不是很简单？一个简易表情标签就这样实现了。

这里再给出一个表达式实现同样的效果：

```
<？php
$str=′ [url]1.gif[/url][url]2.gif[/url][url]3.gif[/url]′；
$s=preg__replace（"#\[url\]（.*？）\[\/url\]#", "<img
src=http：//image.ai.com/upload/$1>", $str）；
var__dump（$s）；
```

提示　这个正则表达式涉及贪婪/懒惰匹配知识，后面会进一步介绍。

3.3.7 环视

断言用来声明一个应该为真的事实。正则表达式中，只有当断言为真时才会继续进行匹配。断言匹配的是一个事实，而不是内容。本节介绍四个断言，它们用于查找在某些内容（但并不包括这些内容）之前或之后，也就是一个位置（如\b、^、）应该满足的一定条件（即断言），因此也称为零宽断言。

1.顺序肯定环视（？=exp）

零宽度正预测先行断言，又称顺序肯定环视，断言自身出现位置的后面能匹配表达式exp。

比如，匹配以“ing”结尾的单词前面部分（除了“ing”以外的部分）：

```
\b\w+(?=ing\b)
```

以上表达式查找以下句子时，会匹配“sing”和“danc”：

```
I'm singing while you're dancing.
```

2.逆序肯定环视（？<=exp）

零宽度正回顾后发断言，又称逆序肯定环视，断言自身出现位置的前面能匹配表达式exp。

比如，以re开头的单词的后半部分（除了re以外的部分）：

```
(?<=\bre)\w+\b
```

以上表达式在查找以下句子时匹配“ading”：

```
reading a book
```

假如在很长的数字中，每3位间加1个逗号（当然是从右边加起），可以在前面和里面添加逗号的部分：

```
（（？<=\d）\d{3}）+\b
```

用以上表达式对“1234567890”进行查找，结果是“，234，567，890”。这里的逗号只是匹配需要添加逗号的位置，还没有实际添加逗号。

下面这个例子同时使用这两种断言，匹配以空白符间隔的数字（再次强调，不包括这些空白符）：

```
（？<=\s）\d+（？=\s）
```

前面提到过反义，用来查找不是某个字符或不在某个字符类里的字符。如果只是想要确保某个字符没有出现，但并不想去匹配它时怎么办？例如，如果想查找这样的单词——出现字母q，但是q后面跟的不是字母u。可以尝试这样：

```
\b\w*q[^u]\w*\b
```

以上表达式匹配包含后面不是字母u的字母q的单词。但是如果多做几次测试就会发现，如果q出现在单词的结尾，例如Iraq、Benq，这个表达式就会出错。这是因为[^u]总要匹配一个字符，如果q是单词的最后一个字符，后面的“[^u]”将会匹配q后面的单词分隔符（可能是空格、句号或其他），后面的“\w*\b”将会匹配下一个单词，于是以上表达式就能匹配整个Iraq fighting。

逆序肯定环视能解决这样的问题，因为它只匹配一个位置，并不消费任何字符。现在，解决这个问题如下所示：

```
\b\w*q（？！u）\w*\b
```

3.顺序否定环视（？！exp）

零宽度负预测先行断言，又称顺序否定环视，断言此位置的后面不

能匹配表达式“exp”。例如：

1）匹配3位数字，而且这3位数字的后面不能是数字：

```
\d{3}(?!\d)
```

2）匹配不包含连续字符串abc的单词：

```
\b((?!abc)\w)+\b
```

如果匹配的单词是c开头、t结尾，中间有一个字符，但不能是u（也就是说，整个单词不能是cut），直接用“c[^u]t”就可以了，若中间的字符不能是a或u（也就是说，整个单词不能是cat或cut），则表达式改为“c[^au]t”。

如果认真读过关于排除型字符组的章节的读者肯定会知道，这个表达式能匹配的只是cot之类的单词，因为中间的排除型字符组“[^au]”必须匹配一个字符。可是，如果还想匹配chart、conduct和court怎么办？最简单的想法是：去掉排除型字符组的长度限制，改成“c[^au]+t”。

不幸的是，这样行不通，因为这个表达式的意思是：c和t之间由多于一个“除a或u之外的字符”构成，而chart、conduct和court都包含a或u。

我们发现，其实要否定的是“单个出现的a或u”，而不仅仅是“出现的a或u”，所以才出现这样的问题。要解决这个问题，就应当把意思准确表达出来，变成“在结尾的t之前，不允许只出现一个a或u”。想到这一步，就可以用顺序否定环视（？！……）来解决。表示在这个位置向右，不允许出现子表达式能够匹配的文本，把子表达式规定为“[au]t\b”（最后的“\b”很重要，它出现在t之后，保证t是单词的结尾字母）。有了限制，匹配a和t之间文本的表达式就随意很多，可以用匹配单词字符的简记法“\w”表示，于是整个表达式变成：

```
c(?![au]t\b)\w+t
```

注意　这里出现的并不是排除型字符组“[^au]”，而是普通的字符组[au]，因为顺序否定环视本身已经表示了否定。

进一步思考，整个匹配文本中都不能出现字符串“cat”，要怎么办呢？这个正则表达式应该是：

```
^（?：（?！cat）.）+$
```

即在文本中的任意位置，都不能出现该字符串。

4.逆序否定环视（？＜！exp）

零宽度负回顾后发断言，又称逆序否定环视，可以用（？＜！exp）断言此位置的前面不能匹配表达式exp。例如，前面不是小写字母的7位数字：

```
（?＜![a-z]）\d{7}
```

分析以下表达式，匹配不包含属性的简单HTML标签内的内容：

```
（?＜=＜（\w+）＞）.*（?＝＜\/\1＞）
```

以上表达式最能表现零宽断言的真正用途。（＜？（＼w+）＞）指定前缀为：被尖括号括起来的单词（比如可能是“＜b＞”），然后是“.*”（任意的字符串），最后是一个后缀（？=＜＼/＼1＞）。注意后缀里的“＼/”，用到了前面提过的字符转义；“＼1”则是反向引用，引用的正是捕获的第一组，即前面（＼w+）匹配的内容，如果前缀实际上是“＜b＞”，后缀就是“＜/b＞”。整个表达式匹配的是“＜b＞”和“＜/b＞”之间的内容（再次提醒，不包括前缀和后缀本身）。

总体而言，环视相当于对“所在位置”附加一个条件，难点就在于找到这个“位置”。这一点解决了，环视就没有什么秘密可言了。

3.3.8 贪婪/懒惰匹配模式

当正则表达式中包含能接受重复的限定符时，通常的行为是（在使整个表达式能得到匹配的前提下）匹配尽可能多的字符。例如以下表达式将匹配以a开始，以b结束的最长字符串：

```
a.*b
```

如果用来搜索“aabab”，它会匹配整个字符串“aabab”。这就是贪婪匹配。

有时，需要匹配尽可能少的字符，也就是懒惰匹配。前面给出的限定符都可以转化为懒惰匹配模式，只要在后面加上一个问号。例如“.*？”就意味着匹配任意数量的重复，但是在能使整个匹配成功的前提下使用最少的重复。例如，匹配以a开始、以b结束的最短字符串，正则表达式如下：

```
a.*?b
```

把上述表达式应用于aabab，如果只考虑“.*？”这个表达式，最先会匹配到aab（1～3字符）和ab（第2～3个字符）这两组字符。

为什么第一个匹配是aab（第1～3个字符）而不是ab（第2～3个字符）？简单地说，正则表达式有另一条规则，比懒惰/贪婪规则的优先级更高：最先开始的匹配拥有最高优先权。

常用懒惰限定符如表3-5所示。

表 3-5　懒惰限定符

懒惰限定符代码/语法	描　　述
*?	重复任意次，但尽可能少重复
+?	重复 1 次或更多次，但尽可能少重复
??	重复 0 次或 1 次，但尽可能少重复
{n,m}?	重复 n 到 m 次，但尽可能少重复
{n,}?	重复 n 次以上，但尽可能少重复

懒惰模式匹配原理简单来说，是在匹配和不匹配都可以的情况下，

优先不匹配，记录备选状态，并将匹配控制交给正则表达式的下一个匹配字符。当后面的匹配失败时，回溯，进行匹配。关于回溯以及正则表达式效率等高级内容，可以查阅《精通正则表达式》一书。

在3.3.6节涉及懒惰匹配，把该节的例子稍作更改：

```
<? php
$str=' [url]1.gif[/url][url]2.gif[/url][url]3.gif[/url]' ;
$s=preg__replace ("#\[url\] (.*) \[\/url\]#", "<img src=http: //image.aiyooyoo.om/upload/$1>", $str) ;
var__dump ($s) ;
```

在贪婪模式下，由于匹配表达式是“.*”，即任意字符出现任意次，这个正则表达式会一直匹配[url]后的内容，直到遇到结束条件“[\ /”。匹配结果如图3-3所示。

图　3-3　运行结果

提示　实际开发中，涉及贪婪模式与懒惰模式的地方是很多的。在一定情况下，使用懒惰模式可以减少回溯，提高效率。

3.4 构造正则表达式

在构造和理解正则表达式的过程中，通常都是由简到繁的过程，如果理解正则表达式内部间的关系，就可以把比较复杂的正则表达式拆分成几个小块来理解，从而帮助消化。

3.4.1 正则表达式的逻辑关系

正则表达式之间的逻辑关系可以简单地用与、或、非来描述，如表3-6所示。

表 3-6 正则表达式间的逻辑关系

逻辑关系	描　述
与	在某个位置，某些元素（字符、字符组或者子表达式）必须出现
或	在某个位置，某个元素或许出现，或许不出现；或长度和出现次数不固定，或者是某几个元素中的一个
非	在某个位置，某些元素不出现

通常来说，正则表达式可以看做这三种逻辑关系的组合。下面分析这三种逻辑。

1.与

“与”是正则表达式中最普遍的逻辑关系。一般来说，如果正则表达式中的元素没有任何量词（比如*、？、+）修饰，就是“与”关系。比如正则表达式：

```
abc
```

表示必须同时出现a、b、c三个字符。

连续字符是“与”关系的最佳代表。此外，有些环视结构也可以表达“与”关系。比如顺序肯定环视（？=exp）表示自身出现的位置后面能匹配表达式exp，换而言之，就是在它后面必须出现表达式exp。例如：

```
\w+(?=ing)
```

表示单词的后面必须是ing结尾。

除了顺序肯定环视外，逆序肯定环视也能表达“与”关系。

比如匹配DIV标签里的内容，例如＜div＞logo＜/div＞中的logo，就可用以下正则表达式来匹配：

```
（？＜=＜div＞）.*（？=＜/div＞）
```

“（？＜=＜div＞）”表示自身（即要匹配的部分）出现的位置前面匹配表达式“＜div＞”，“（？=＜/div＞）”表示它的后面需要匹配表达式“＜/div＞”，中间的“.*”就是匹配到的内容。

2.或

“或”是正则表达式中容易出现的逻辑关系。

如果“或”代表元素可以出现，也可以不出现，或者出现的次数不确定，可以用量词来表示“或”关系。比如以下表达式表示在此处，字符a可以出现，也可以不出现：

```
?a?？
```

以下表达式表示在此处，字符串ab必然要出现1次，也可以出现无限多次：

```
『（ab）+』
```

如果“或”表示出现的是某个元素的一个，那么可以使用字符组。比如以下正则表达式表示此处出现的字符是a、b、c中的任何一个：

```
[abc]
```

如果要匹配多个字符，则使用分支结构（……丨……）。比如匹配单词foot及其复数形式，就可以用正则表达式：

```
f（oo|ee）t
```

或者使用以下形式：

```
f[oe]{2}t
3.非
```

提到“非”，最容易想到正则表达式中的反义和“^”元字符。比如“\d”表示数字，那么其对应的\D就表示非数字；[a]表示a字符，那么[^a]就表示这个字符不是a。

“非”关系最常用来匹配成对的标签，例如双引号字符串的匹配，首尾两个双引号很容易匹配，其中的内容肯定不是双引号（暂不考虑转义的情况），所以可以用[^"]表示，其长度不确定，用*来限定，所以整个表达式如下：

```
[^"]*
```

比如，需要匹配HTML里成对的A标签，先匹配左尖括号，紧接着是a字符，后面可以是任意内容，最后是一个右尖括号。在这对括号之间可以出现空格、字母、数字、引号等字符，但是不能出现“>”字符，于是就可以用排除型字符组“[^>]”来表示。再加上后面的配对标签，整个表达式如下：

```
<a[^>]*>.*<\/a>
```

运行下面这段代码验证这个表达式：

```
<?php
$reg="#<a[^>]*>（.*）<\/a>#";
$str='<a href="http：//baidu.com">baidu</a>some<a href="http：//sohu.com">sohu</a>';
preg_match_all（$reg，$str，$m）；
var_dump（$m）；
```

运行结果如下：

```
array（2）{
[0]=>
array（1）{
[0]=>
string（74）"<a href="http：//baidu.com">baidu</a>some<a href="http：//sohu.com">
sohu</a>"
}
```

```
[1]=>
array（1） {
[0]=>
string（43） "baidu</a>some<a href="http: //sohu.com">sohu"
}
}
```

发现结果不符合预期，出现了嵌套匹配。原因在于A标签之间的文本忘了做排除型匹配，于是修改后的正则表达式就成了“<a[^ >]*>（[^ <>]*）<\/a>”。经过修改后就符合预期了。

除了反义和排除型字符组外，否定环视也能表示“非”这种关系。比如有一串文字：“ab<p>onecde fgh <img src="">”。现在需要匹配除P标签外的所有标签。换而言之，就是先匹配所有HTML标签，可以使用如下表达式：

```
</? \b[ ^ >]+>
```

匹配闭合的“<XXX>”或“</XXX>”标签，然后再排除“XXX”或“/XXX”部分是P的标签，于是使用顺序否定环视，用表达式：

```
（? ! /? p\b）
```

排除了“<”或“</”这个位置后是P字符的情况，这样就满足需求了。最终的表达式则为：

```
<（? ! /? p\b）[ ^ >]+>
```

通过上面的分析得到正则表达式中的“与或非”关系及其代表语法，如表3-7所示。

表 3-7　正则表达式中的“与或非”关系及其代表语法

逻辑关系	代表语法
与	连续字符、肯定环视（顺序肯定，逆序肯定）
或	量词、字符组
非	排除型字符、反义、否定环视（顺序否定，逆序否定）

3.4.2 运算符优先级

正则表达式从左到右进行计算，并遵循优先级顺序，这与算术表达式非常类似。表3-8说明了各种正则表达式运算符的优先级顺序，其中优先级从上到下、由高到低排列。

表 3-8 正则表达式运算符的优先级

运 算 符	描 述
\	转义符
(), (?:), (?=), []	括号和中括号
*, +,?, {n}, {n,}, {n, m}	限定符
^, $, \anymetacharacter, anycharacter	定位点和序列
|	替换

字符的优先级比替换运算符高，替换运算符允许m｜food与m或food匹配。要匹配mood或food，使用括号创建子表达式，从而产生如下表达式：

```
(m|f)ood
```

3.4.3 正则表达式的常用模式

模式（Pattern Modifiers）就是可以改变表达行为的字符，用来关闭或打开某些特殊功能，习惯上又称为正则修饰符。每种语言提供的正则表达式所支持的修饰符都不一样，这节介绍一些基本修饰符及常用模式。

1.忽略大小写模式（i）

在此模式下，正则匹配将不区分待匹配内容的大小写，这在HTML里常用。由于HTML本身的容错性很好，对大小写混用有很好的兼容处理能力，也就经常会出现无论是标签还是内容的大小写混乱问题，这时采用这种模式就能很好地处理这种情况。示例代码如下：

```
<div>gg<\/div>在忽略大小写模式下，可匹配：<div>gG</Div>
<? php
$str=' <div>gG</Div>' ;
if（preg_match（' %<div>gg<\/div>%i' , "<div>gG</Div>", $arr）） {
echo"匹配成功".$arr[0]；
} else {
echo"匹配不成功"；
}
```

忽略大小写是针对整个表达式而言，而不仅仅是欲匹配的部分。所以，可以在代码里放心地使用此修饰符。但是出于效率的考虑，尽量让正则表达式所指示的范围更精确。

注意，修饰符对整个表达式有效。如果只想修饰部分表达式，可以使用PCRE的内部选项——“局部修饰符”。比如下面表达式仅匹配abc和abC，而不会匹配Abc：

```
#ab（? i）c#
```

也就是说，（？i）只对它后面的字符c起作用。

2.多行模式（m）

在讲起始和终止符时提过：“勾选Multiline选项，即多行选项。如果选中了这个选项，“ ^ ”和“$”的意义就变成匹配行的开始处和结束处，否则将把整个输入视作一个字符串，忽视换行符。”也就是说，正则表

达式默认开始 ^ 和结束只是对于正则字符串，如果在修饰符中加上m，开始和结束将会指字符串的每一行：即每一行的开头就是 ^ ，结尾就是$。

需要注意，m表示多行匹配，而非跨行匹配。仅当表达式中出现 ^ 、中至少一个元字符且字符串有换行符“\n”时，m修饰符才起作用，否则被忽略，如代码清单3-1所示。

代码清单3-1　多行模式

```
<? php
$str="this is reg
Reg
this is
regexp turtor, oh reg";
if（preg_match_all（' %.*reg$%mi' , $str, $arr）） {
echo"匹配成功";
var_dump（$arr）;
} else {
echo"匹配不成功";
}
```

在预想中，“.*reg $ ”就是以reg结尾的行，由于加了m修饰符，按理应该匹配第一行和第四行，但实际结果呢？如下所示：

```
匹配成功array（1） {
[0]=>
array（1） {
[0]=>
string（20）"regexp turtor, oh reg"
}
}
```

可以看出，只匹配最后一行的reg，而第一行虽然也是以reg结尾，但是并没有被匹配。这里用到mi，也就是把修饰符m和i组合使用。事实上，本例中即使去掉m修饰符，最终结果也是一样，这说明 $ 只能表示最后一行。把正则表达式改为：

```
%^t.*%mi
```

匹配的结果将是：

```
匹配成功array（1） {
[0]=>
array（2） {
[0]=>
string（12）"this is reg"
[1]=>
string（8）"this is"
}
}
```

匹配到两行，去掉m修饰符只匹配到第一行。可见，即使加了m修饰符，也不是将整个字符串都匹配，这就是跨行与多行的区别。

另外，使用m模式匹配需要注意换行符是否真的有效。看代码清单3-2所示的例子。

代码清单3-2　多行模式的例子

```
<? php
$source1=' abc\nabcd' ;
$source2="abc\nabcd";
if(preg_match_all(' %^abc%m' , $source1, $arr) ) {
echo"匹配成功——";
var_dump($arr);
} else {
echo"匹配不成功——";
}
if(preg_match_all(' %^abc%m' , $source2, $arr) ) {
echo"匹配成功——";
var_dump($arr);
} else {
echo"匹配不成功——";
}
```

运行可以看到，由于source1字符串使用单引号，\n作为普通字符而非换行符，因此匹配到的结果和预期不符。

3.点号通配模式（s）

点号通配模式的作用是使正则表达式里的点号元字符可以匹配换行符，如果没有这个修饰符，点号不匹配换行符。沿用上面的例子，如代码清单3-3所示。

代码清单3-3　点号通配模式

```
<? php
$str="this is reg
Reg
this is
regexp turtor, oh reg";
if(preg_match_all(' %this.*? reg%i' , $str, $arr) ) {
echo"匹配成功";
var_dump($arr);
} else {
echo"匹配不成功";
}
```

前面学习过，“.”元字符表示除换行符以外的任意字符。所以按照这个推论，上述正则表达式应该能匹配到第一行，即“this is reg”，而第三行和第四行由于之间存在一个换行符，所以不能匹配。如果给上述正则表达式加上s修饰符，即“%this.*？reg%is”，那么匹配结果就不同了，如

下所示：

```
匹配成功array（1） {
[0]=>
array（2） {
[0]=>
string（11） "this is reg"
[1]=>
string（13） "this is reg"
}
}
```

可以看出，这个匹配跨行了。

我们只要牢记，s修饰符包括换行符。这个修饰符很有用，特别是在抓取一些文档时，由于存在不可见换行（这是很常见的），如果使用“.”匹配，就可能存在问题，这就需要表达式能匹配换行符。看代码清单3-4所示的例子。

代码清单3-4　匹配换行符

```
$str=′ <body>
<div class="content">
<div class="head"></div>
<div class="body"></div>
<div class="foot"></div>
</div>
</body>′；
$array=array（）；
$array2=array（）；
preg_match_all（′ |<body>（.*）<\/body>|′ , $str, $array）；
var_dump（$array）；
preg_match_all（′ |<body>（.*）<\/body>|s′ , $str, $array2）；
var_dump（$array2）；
```

结果是第一个正则表达式没有匹配到任何内容，而第二个正则表达式匹配＜body＞标签之内的全部内容。原因在于，要匹配的文本在＜body＞标签后紧接着是一个不可见换行符，从而导致第一个正则表达式匹配失败。从这个例子中更能深刻体会到s修饰符的作用。

4.懒惰模式（U）

“U”相当于前面提到的“？”，表示懒惰匹配，因此3.3.8节的例子也可以改写为如下代码：

```
<？php
$str=′ [url]1.gif[/url][url]2.gif[/url][url]3.gif[/url]′；
$s=preg_replace（"#\[url\]（.*）\[\/url\]#U", "<img src=http：//image.aiyooyoo.om/upload/$1>", $str）；
var_dump（$s）；
```

在这里：以下两个表达式作用是等价的。

```
#\[url\](.*)\[\/url\]#U
```

和

```
#\[url\](.*?)\[\/url\]#
```

5.结尾限制（D）

如果使用限制结尾字符，则不允许结尾有换行。例如，以下正则表达式将匹配“abc”、“abc＼n”这样的字符，即忽视结尾的换行：

```
%abc$%
```

如果使用此模式，限定其不可有换行，必须以abc结尾，如下所示：

```
%abc$%D
```

6.支持UTF-8转义表达（u）

u修饰符启用PCRE中与Perl不兼容的额外功能，模式字符串被当成UTF-8。u修饰符在UNIX下自PHP 4.1.0起可用，在Win32下自PHP 4.2.3起可用，自PHP 4.3.5起开始检查模式的UTF-8合法性。看一个例子：

```
$str="php编程";
if(preg_match("/^[\x{4e00}-\x{9fa5}]+$/u",$str)){
echo"该字符串全部是中文";
}else{
echo"该字符串不全部是中文";
}
```

正则表达式中使用了u修饰符，就可以使用UTF-8的转义表达，这实际上是兼容问题。在PHP中，使用此修饰符即可实现在表达式中支持UTF-8。

提示　除了上述几个模式修饰符之外，PHP里还支持另外几个修饰符，如A、x等，但不是很常用，这里就不做讲解了。

3.5　正则在实际开发中的应用

前面几节主要讲解了正则表达式一些基础知识，本节几个例子将展示正则表达式在日常开发中的应用。

3.5.1　移动手机校验

首先，我们练习最常用的正则表达式：移动手机校验。

通过查阅相关资料知道移动的号段有：134、135、136、137、138、139、150、151、157、158、159，新增3G号182、183和188。手机号码一共11位，前3位为运营商号段，中间4位为号码归属地，后4位为随机号。那么可以写出如下正则表达式：

```
（13[4-9]|15[01789]|18[238]）\d{8}
```

规则很简单，匹配前3位是否在移动号码段里，后8位为数字即可。这里用到分支、分组和元字符这三个知识点，用PHP代码测试如代码清单3-5所示。

代码清单3-5　测试移动手机号匹配

```
<? php
$mobile=' 13500000000' ;
$regex="! ^（13[4-9]|15[0189]|188）\d{8} $! ";
if（! preg_match（$regex, $mobile））{
die（' 错误的移动号码段! ' ）;
} else {
die（' 移动手机.' ）;
}
```

运行结果符合预期。注意第三行代码，在前面已经说过，正则表达式的分隔符只要遵循规则是可以随意的。所以这里用的是感叹号，而不是常见的斜杠，当然，用#、～也可以（为了书写美观，一般不用斜杠）。

如果只判断手机号就更简单了，如下所示：

```
1[358]\d{9}
```

3.5.2 匹配E-mail地址

运用学过的知识，我们实现简单的E-mail识别。

E-mail最简单的形式为user@domain.com。其中，user为用户名，domain为域名，com为后缀；当然，后缀还可以是net、name、cn等。一般用户名由3个以上的字母和数字组成，当然也不能太长，允许出现下画线。中间一个@符号，后面的域名长度为1～64位，后缀长度一般为2～5位，如下所示：

```
\w{3, 16}@\w{1, 64} \.\w{2, 5}
```

以上表达式匹配用户名长度3～16位，紧跟一个@符号，然后是1～64位的域名，再然后是dot（.号），最后是2～5位的后缀。这个正则表达式还存在一些问题，比如xxx__yyy@ya hoo.com.cn这样的地址，后面的.cn就无法匹配到。这就需要进一步学习。

Regular Expression Tester工具提供一些预定义的正则表达式，它提供的匹配E-mail的正则表达式如下：

```
^[a-z0-9_\-]+(\.[_a-z0-9\-]+)*@([_a-z0-9\-]+\.)+([a-z]{2}|aero|arpa|biz|com|coop
|edu|gov|info|int|jobs|mil|museum|name|nato|net|org|pro|travel)$
```

这个表达式很长，使用字符组和分支条件语法，很好地解决了com.cn这样多个后缀域名的问题。相信现在读者对于匹配电话号码、QQ号等应该能得心应手了。

3.5.3 转义在数据安全中的应用

数据安全是任何一款软件设计中都需要考虑的问题。从技术层面讲，数据安全就是保护存储和使用的数据不被窃听、盗取和破坏。这可能是由外部因素造成的，比如由于过滤不严格造成SQL注入漏洞、提升脚本执行权限等，也有可能是由代码内部设计造成的，如死循环、低效率的语句造成服务器性能下降以致影响访问。

社会学意义上的数据安全则更广泛。比如，在在线购物商城的设计中，由于设计者错误地使用自增ID作为商品的单据流水号，竞争对手或有心人很容易分析出这个商城的每日销售量，进而估算出销售额、利润等商业机密数据。

在程序中要保证数据的安全，除了要保证代码内部运行的可靠外，最主要的就是严格处理外部数据，即秉持“一切输入输出都是不可靠的”理论，这就要求我们做好数据过滤和验证。PHP编程中最简单的过滤机制就是转义，即对用户的输入和输出进行转义和过滤。图3-4所示是简单的留言本。

简单留言本的演示代码如下：

```
<form action=""method="POST">
<textarea name="content"></textarea>
<input type="submit"value="留言"/>
<? php
echo $_POST[' content' ];
? >
```

图 3-4 简单的留言本

程序中没有任何的输入输出过滤，当在留言框中输入以下内容时得到如图3-5所示的页面。

```
<style>
Body{background: #000000}
```

```
</style>
```

很显然，由于输入含有CSS代码和JavaScript代码，没有对其进行处理便原样输出，导致页面被篡改。这就是常说的“XSS攻击”。

针对上面的情况，只需在接收数据时使用htmlspecial chars函数，把代码中的特殊字符转为HTML实体，这样在输出时就不会使页面受影响了。这些特殊字符主要是“"”、“'”、“&”、“<”、“>”。比如把“<script>alert（1）；</script>”转为“&lt；script&gt；alert（1）；&lt；/script&gt；”，就可以阻止大部分XSS攻击。

图　3-5　被“攻击”后的留言本

注　HTML中规定字符实体引用（HTML Entities）还有对应的数字型实体（NCR），实际上是对这个实体的编号。比如，HTML Entities的格式“&lt；”，NCR的格式“&#60；”或“&#x3c；”，均表示“<”字符。

有的时候，不希望出现这些无意义的字符，因为既然页面不允许这些HTML标签，那就干脆过滤掉，而不是显示出来，这样页面就不会留下恶意攻击者所留下的这些代码。要达到这个目的，就需要借助正则表达式。比如，要过滤所有HTML标签的正则表达式如下所示：

```
<\/? [^>]+>//过滤所有HTML标签
```

这个正则表达式匹配嵌套的尖括号，一个“\/？”表示斜杠可有可无，这样就匹配标签的起始和关闭位置。“[^>]+”意思是：不是右尖括号的字符重复一次和更多次。为什么不是“*”呢？因为HTML标签里至少也是一个字符，如<b>，而<>不是合法HTML标签，最后关闭右尖括号。用下面字符测试：

```
<strong>strong</strong><img src=a.jpg/>
```

其匹配情况如图3-6所示。

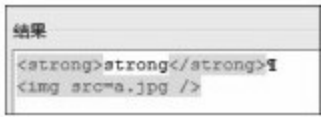

图　3-6　程序运行结果

从图中可以看出，运行结果基本符合我们的需求。同理，还可以只保留部分HTML标签，既不造成安全问题，又能使页面内容更丰富，这就可以利用UBB代码功能来实现。

前面提到过用正则表达式实现UBB标签的功能比较麻烦，但在这里只需要白名单功能，即只保留比较“安全”的标签，通过PHP内置的strip__tags函数可以容易做到。这个函数用于从字符串中去除HTML和PHP标记，仅保留参数中指定的标签。演示代码如下所示：

```
<? php
$text=' <p>Test paragraph.<! —Comment—><a href="#fragment">Other text</a>' ;
echo strip__tags（$text）;
echo"\n";
//允许<p>和<a>
echo strip__tags（$text, ' <p><a>' ）;
? >
```

另外，在SQL语句构造中，当字符含有引号的时候，可能造成SQL解析和执行失败。这样就要求转义。一种处理办法是把引号转义成实体，另一种处理办法是使用addslashes函数把其转义。我们通常会使用后者。

addslashes函数返回一个字符串，该字符串为了数据库查询语句等的需要在某些字符前加上了反斜线。这些字符是单引号（′）、双引号（"）、反斜线（\）与NUL（NULL字符）。

注意　为了处理更多情况，还需要特别注意“%”、“′”等特殊字符。如果能转义则进行转义，没有对应的转义时，则进行过滤。

3.5.4 URL重写与搜索引擎优化

URL重写（Rewrite）是截取传入Web请求并自动将请求重定向到其他URL的过程。比如浏览器发来请求hostname/list__1，服务器自动将这个请求中定向为http：//hostname/list.php？id=1。该技术常用于搜索引擎优化（Search Engine Optimization, SEO）。

伪静态也是重写的一种实现。例如，某论坛网址为forum 17 1.html，实际地址可能是fo rum.php？fid=17＆page=1，其中第一个数字是版块ID，第二个数字是页数。这样的网址看起来比较短，没有一大堆的＆符号和GET参数，不但用户喜欢这种友好的网址，搜索引擎也喜欢这种简洁明了的网址。

URL重写有两种实现方式：

纯代码实现，通过解析PATH__INFO实现。

服务器实现，如利用Apache的mod__rewrite模块实现。

下面先介绍利用mod__rewrite实现重写的方法。例如，实现列表页面list.php？Mode=A重写为伪静态list A.html，步骤如下（假设工程的根目录是E：/dev/www/php/new）。

首先，配置Apache。在httpd.conf里把以下所示的这一行代码前面的#去掉，启用Rewrite规则。

```
#LoadModule rewrite__module modules/mod__rewrite.so
```

在对应的＜Directory"E：/dev/www/php"＞配置项下设置AllowOverride All。

在网站E：/dev/www/php/new目录下新建.htaccess文件，并输入如下内容：

```
RewriteEngine on
RewriteRule index.html index.php
RewriteRule list-（[A-Z]+）\.html$list.php？mode=$1[NC]
```

重启Apache。现在访问http：//127.1/list A.html看看效果吧。可以看到访问http：//127.1/list A.html和http：//127.1/list.php？mode=A指向的是同一个页面，这就说明伪静态设置成功了。

现在逐条解释其原理。

Apache开启Rewrite模块，该模块默认是关闭的。

设置AllowOverride All的目的是让.htaccess文件生效；如果把AllowOverride设置成none，.htaccess文件将被完全忽略。

新建一个.htaccess文件。.htaccess文件是Apache中重要的配置文件，其格式为纯文本。它提供针对目录改变配置的方法，通过在一个特定的文档目录中放置包含一个或多个指令的文件，作用于此目录及其所有子目录。

注意 Windows资源管理器里不允许建立.htaccess文件，可以在命令行窗口输入“echo＞.htaccess”达到新建的目的。

以下是对.htaccess中每一行代码的解释。

1）打开运行时重写功能，其默认是关闭：

```
RewriteEngine on
```

2）建立一条重写规则，把index.php重写为index.html：

```
RewriteRule index.html index.php
```

很显然，这是一个伪静态，而“index.html index.php”是最简单的正则表达式。

3）把原地址list.php？mode=1形式重写成list （[A-Z]+）＼.html：

```
RewriteRule list-（[A-Z]+）＼.html$list.php?mode=$1[NC]
```

实际上上述代码可以理解为，如果Apache遇到符合正则表达式list

（[A-Z]+）＼.html的请求，先捕获第一个括号里的值（这是紧挨着list后面，以.html结尾的一个字母），那么就重定向到真实的请求地址list.php？mode=1，而1是正则表达式里的反向引用。这就是重写的语意。括号中的“NC”表示大小写不敏感。

如果这个页面还有分页应该怎么写？比如list.php？mode=A＆page=2。为了让URL更有意义，可以把这个地址重写为list A page 2.html。照葫芦画瓢，在已有基础上添加一条规则：

```
RewriteRule list-（[A-Z]+）-page-（＼d？）＼.html＄list.php？mode=＄1＆page=＄2
```

现在直接访问list A page 2.html试试。由于RewriteEngine为on，所以这一步并不需要重启Apache。虽然现在可以访问了，但是不是觉得这么写很累赘？

第二条和第三条重写规则都是针对list.php文件，可以合并成一条吗？答案是可以的。完整.htaccess文件如下：

```
RewriteEngine on
RewriteRule index.html index.php
#RewriteRule list-（[A-Z]+）＼.html＄list.php？mode=＄1[NC]这一行是注释
RewriteRule list-（[A-Z]+）（？：-page-）？（＼d？）＼.html＄list.php？mode=＄1＆page=＄2[NC]
```

新的重写规则就是最常用的正则表达式，如“？：”表示非捕获性匹配。只要理解了正则表达式，再辅之以Apache手册，掌握URL重写就会成为一件很轻松地事情。

提示　Nginx也能实现网址静态化，而且语法上几乎没什么差别，用到的都是基于正则表达式的语法。

使用URL重写能给网站带来哪些好处呢？

1）有利于搜索引擎的抓取。因为现在大部分搜索引擎更喜欢抓取一些静态页面。而现在页面大部分数据都是动态显示的。这就需要把动态页面变成静态页面，这样有利于搜索引擎抓取。

2）用户更容易记忆。很少有用户关心网站页面地址，但对大中型网站来说，增强可读性还是必需的。

3）隐藏实现技术。避免暴露采用的技术，给攻击网站增加阻碍。特别是前面讲的攻击方式，把参数隐藏起来，在一定程度增加了攻击的难度。

重写还能帮助我们完成很多事情，比如简单下载处理。例如，现在需要对外提供文件下载服务，比如下载php.rar文件。在下载前还需要做许多额外工作，如下载权限的判断、服务器分流等，这就不是一个简单文件链接可以处理的，需要服务器端脚本做一些复杂处理。通过使用URL重写，可以把对php.rar文件的请求重定向到对一个PHP文件的请求，在这个PHP文件中进行判断，满足所有判断后再输出文件。假定当前工程目录是E：＼www＼php＼download，现在要把对E：＼www＼php＼download＼php.rar的请求定向到一个PHP文件。添加下面的代码到当前目录的.htaccess文件中：

```
RewriteEngine on
RewriteRule（.*＼.（rar｜zip｜chm））＄down.php?file=＄1[NC]
```

down. php中的代码如下：

```
＜? php//echo"The file you wanna-download is＼t"，＄__GET[′file′]；
＄filename=basename（＄__GET[′file′]）；
//其他逻辑处理，如检查是否有权限或者是否属于盗链，处理完成后提供下载
if（! is__file（＄filename）｜｜! is__readable（＄filename））｛
exit（"没有找到指定的文件"）；
｝
header（"Content-type：application/octet-stream"）；
＄ua=＄__SERVER["HTTP__USER__AGENT"]；
//处理中文文件名
＄encode__filename=urlencode（＄filename）；
＄encode__filename=str__replace（"+"，"%20"，＄filename）；
if（preg__match（"/MSIE/"，＄ua））｛
header（′Content-Disposition：attachment；filename="′.＄encode__filename.′"′）；
｝else if（preg__match（"/Firefox/"，＄ua））｛
header（"Content-Disposition：attachment；filename*=＼"utf8′′".＄filename.′"′）；
｝else｛
header（′Content-Disposition：attachment；filename="′.＄filename.′"′）；
｝
//如果服务器支持X__sendfile moudle，则使用X__sendfile头加速下载
＄x__sendfile__supported=in__array（′mod__xsendfile′，apache__get__modules（））；
if（! headers__sent（）＆＆＄x__sendfile__supported）｛
header（"X-Sendfile：｛＄filename｝"）；
｝else｛
@readfile（＄filename）；
｝
```

再浏览http：//download.com/hello.rar（已将虚拟主机域download.com映射到E：＼www＼php＼

download目录），此请求将被重定向到down.php？file=hello.rar，这样就简单的实现了对文件下载的处理。

我们还能对重写做进一步扩充，来满足不同的需求。更多URL重写

知识可以参考Apache 2手册。

3.5.5 删除文件中的空行和注释

不仅在代码中会用到正则表达式，其实在日常软件应用中也会涉及正则表达式。比如字处理软件、代码开发工具中都提供对正则表达式查找和替换的支持。

这里以UltraEdit为例来介绍正则表达式在日常软件中的应用。UltraEdit是一款功能强大的编辑器，支持正则表达式的使用。UltraEdit虽然和IDE无法相提并论，但是在处理一些小文件时，会显出其快速、轻量级的特点。

例如，PHP源文件中包含空行和注释，UltraEdit中的代码如图3-7所示。

这里面许多空行和注释，为了提高代码的可读性，需要去除大段空行。如果手工操作，必然很麻烦。此时，可以使用UltraEdit的正则表达式功能。在编辑菜单中，选择“替换”，输入如下表达式：

```
%[^t]++^p
```

注意，^t前面的空格也要输入。单击替换所有，文件中的空行就删除了。如果还要删除注释，可以输入“//？*”，处理完成后的效果如图3-8所示。

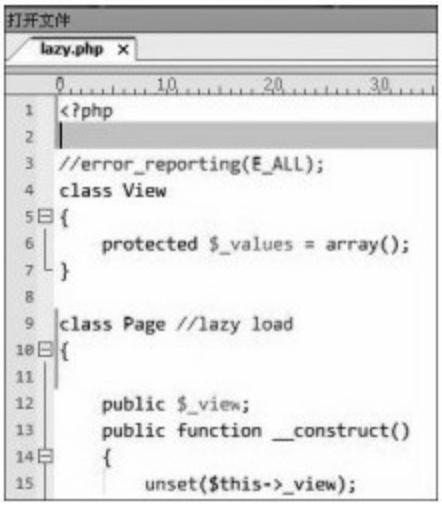

图 3-7 UltraEdit中的代码

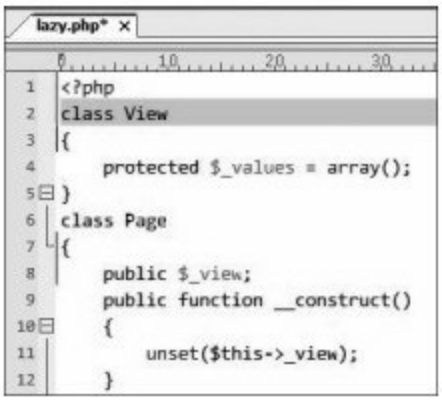

图　3-8　程序运行结果

这里使用UltraEdit的正则表达式，也可以选择UNIX（POSIX规范）和Perl（PCRE规范）风格的表达式，它们之间略有不同。

提示　有些框架为了尽力提升效率或者由于商业的原因，往往会在部署和发布时，通过解析PHP代码中的token清除源文件中表示空白和注释的token。在这种情况下，使用代码的方式可能更好。

但有时无法使用代码完成这件事，我们不得不使用正则表达式。比如在使用Word保存资料的时候，文件中常常会带有大量的空白段落，通常只能手动删除这些空段落来调整格式，费时费力。在Word中，选择特殊字符，把 ^ p ^ p替换成 ^ p即可。Word中这两个所谓的“特殊字符”，实际上就是正则表达式的一种体现。

3.6　正则表达式的效率与优化

正则表达式可以看做描述字符串匹配的算法代码，本质上说是一种有限状态机在计算机中的表示方法。状态机，表示有限个状态以及在这些状态之间进行转移和动作等行为的数学模型，在计算机中表示出来的就是有向图。而作为图就会涉及查找、回溯过程。不同查找的方式，其回溯过程也不一样，效率自然也是有区别的。要想弄明白正则表达式的效率，就得深入编译原理、数据结构等概念，这里不做原理性阐述，只介绍一些普遍原则。

注意　不同语言中有不同实现和限制，因此下面一些原则只是最基本的原则，不保证在所有实现中通用。有时候，某一种实现会对预知的情况进行优化，而另一种则不会。也就是说，正则表达式的效率不仅和正则引擎的种类有关，还和引擎具体实现有关。

1）使用字符组代替分支条件。比如，使用[a-d]表示a～d之间的字母，而不是使用（a｜b｜c｜d）。下面代码说明二者的效率差异。

```
<? php
$cnt=1000;
$testStr="";
for ($i=0; $i<1000; $i++) {
$testStr.="abababcdefg";
}
//第一种方案
$start=microtime (TRUE);
for ($i=0; $i<$cnt; $i++) {
preg_match ('#^ (a|b|c|d|e|f|g)+$#', $testStr);
}
echo' waste time (s): ', microtime (TRUE) -$start, PHP_EOL;
//第二种方案
$start=microtime (TRUE);
for ($i=0; $i<$cnt; $i++) {
preg_match ('#^[a-g]+$#', $testStr);
}
echo' waste time (s): ', microtime (TRUE) -$start, PHP_EOL;
//第三种方案和第二种方案本质上是相同的
$start=microtime (TRUE);
for ($i=0; $i<$cnt; $i++) {
preg_match ('#^[abcdefg]+$#', $testStr);
}
echo' waste time (s): ', microtime (TRUE) -$start, PHP_EOL;
```

运行结果如下所示：

```
waste time (s): 3.2742960453033
waste time (s): 0.059613943099976
waste time (s): 0.059823036193848
```

可以看出，[a　g]和[abcdefg]这两种表达式的效率相当，且使用字符组比分支条件的速度要快很多。这是由于在匹配单个字符的时候，引擎

会把[abc]这样的字符组视为1个元素，而不是3个元素（a、b、c）。整个元素作为匹配迭代的一个单元，不需要进行三次迭代，从而提高匹配效率。

2）优先选择最左端的匹配结果。这在介绍分支条件匹配邮编的时候已经提到过。对于传统型NFA引擎来说，这样改动对正则匹配的效率是有利的，因为引擎一旦找到匹配结果就会停下来，而不会去尝试正则表达式的每一种可能（PHP中的preg函数就属于传统型NFA引擎）。

3）标准量词是匹配优先的。若用量词约束某个表达式，那么在匹配成功前，进行的尝试次数有下限和上限。例如，正则表达式为：

```
\w*（\d+）
```

字符串为copy2003y。这个正则匹配的1是多少？如果回答2003就错了，其结果应该是3。解释如下：当正则引擎用“＼w*（＼d+）”匹配字符串copy2003y时，会先用“＼w*”匹配字符串copy2003y。“＼w*”会匹配字符串copy2003y的所有字符，然后再交给“＼d+”匹配剩下的字符串，而剩下的没有了。这时，“＼w*”规则会不情愿地吐出一个字符，给“＼d+”匹配。同时，在吐出字符之前，记录一个点。这个点就是用于回溯的点，然后“＼d+”匹配y，发现不能匹配成功，此时会要求“＼w*”再吐出一个字符；“＼w*”先记录一个回溯的点，再吐出一个字符。这时，“＼w*”匹配结果只有copy200，已经吐出3y。“＼d+”再去匹配3，发现匹配成功，会通知引擎，并且直接显示出来。所以，“（＼d+）”的结果是3，而不是2003。

如果改为非贪婪模式呢？“＼w*？（＼d+）”匹配结果就应该是2003。由于“＼w*？”是非贪婪，正则引擎会用表达式“＼w+？”每次仅匹配一个字符串，然后再将控制权交给后面的“＼d+”匹配下一个字符，同时记录一个点，用于匹配不成功时，返回这里再次匹配。

提示　尽量以组为单位进行匹配，使用固化分组就能避免无休止的匹配。

4）谨慎用点号元字符，尽可能不用星号和加号这样的任意量词。只要能确定范围（例如“＼w”），就不要用点号；只要能够预测重复次数，就不要用任意量词。假设一条微博消息的XML正文部分结构如

下：

```
<span class="msg">……</span>
```

正文中无尖括号，写法如下：

```
<span class="msg">[^<]{1，200}</span>
```

或者：

```
<span class="msg">.*</span>，
```

上述第一种代码的思路要好于第二种代码，原因有两个：

使用“[^ <]”，保证了文本的范围不会超出下一个小于号所在位置。

明确长度范围｛1，200｝，依据是一条微博消息大致的字符长度范围是固定的，现在微博字数长度限制是140个字。

PHP的PCRE扩展中提供了两个设置项：

pcre. backtrack__limit//最大回溯数

pcre. recursion__limit//最大嵌套数

默认backtarck__limit是100000（10万），recursion__limit限制最大正则嵌套层数。在正则表达式的使用中，应尽量避免回溯次数过多等情况。

因回溯次数过多导致正则匹配失败的案例如下：

```
<? php
$a=range（1，12636）；
shuffle（$a）；
$d=print__r（$a, TRUE）；
echo strlen（$d）/1024，PHP__EOL；
$a="<? xml version=′ 1.0′ encoding=′ iso-8859-1′ ? ><ppc>header".$d."tail</ppc>"；
preg__match__all（"/<ppc>（.*? ）（\d*）<\/ppc>/s"，$a，$m，PREG__SET__ORDER）；
var__dump（$m）；
echo′ had result：′ ，PHP__EOL；
echo
strlen（$m[0][1]），PHP__EOL，substr（$m[0][1]，0，50），PHP__EOL，substr（$m[0][1]，-50），
PHP__EOL；
```

```
//复制上面的代码，唯一的修改是增加字符串长度，使正则匹配爆栈
//ini_set（' pcre.backtrack_limit' , 100000000）; //这里增加为1亿
$x2=range（1，12638）;
shuffle（$x2）;
$d2=print_r（$x2，TRUE）;
echo strlen（$d2）/1024，PHP_EOL;
$a2="<? xml version=' 1.0' encoding=' iso-8859-1' ? ><ppc>header".$d2."tail</ppc>";
$ret=preg_match_all（"/<ppc>（.*? ）（\d*）<\/ppc>/s", $a2，$m2，PREG_SET_ORDER）;
var_dump（$m）;
echo preg_last_error（）;
echo' had no result: ' , PHP_EOL;
echo
strlen（$m2[0][1]）, PHP_EOL, substr（$m2[0][1]，0，50）, PHP_EOL, substr（$m2[0][1]，-50）;
```

注意　程序的运行结果依赖于你的PHP的设置。

这个案例告诉我们，由于正则表达式使用不当导致匹配失败的情况是有可能发生的，特别是当Web文档比较大、结构比较复杂时。解决办法就是，把以下代码前面的注释符去除，给PHP配置更大的回溯栈空间：

```
ini_set（' pcre.backtrack_limit' , 100000000）;
```

但这是治标不治本的办法，最终解决方案还是优化正则表达式。

同理，能用懒惰匹配就坚决不用贪婪匹配。

5）尽量使用字符串函数处理代替。使用字符串函数和正则表达式都可以处理字符串，两者相比，字符串函数处理的效率更高。当然，有些情况几乎是非正则表达式不能胜任的，或者不用正则表达式的成本太高，这些情况不得不用正则表达式，既然如此，就应该设计好。

6）合理使用括号。每使用一个普通括号（），而不是非捕获型括号（？：.....），就会保留一部分内存等着再次访问。这样的正则表达式、无限次的运行次数，无异于一根根稻草的堆加，终将会把骆驼压死。

7）起始、行描点优化。能确定起始位置，使用 ^ 能提高匹配的速度。同理，使用标记结尾，正则引擎则会从符合条件的长度处开始匹配，略过目标字符串中许多可能的字符。在写正则表达式时，应该将描点独立出来，例如“ ^ （？：abc | 123）比 ^ 123 | ^ abc”效率高，而“ ^ （abc）比（ ^ abc）”效率要高。

这个原则不适用于所有正则引擎。比如在PCRE中，二者效率相当。

8）量词等价转换的效率差异。例如在PHP中，使用“\d\d\d”和“\d{3}”，或者“====”和“={4}”，它们之间的效率几乎没有差别。但是换用其他语言可能就会有比较明显的性能差异了。

9）对大而全的表达式进行拆分。

10）使用正则以外的解决方案。前面已经提到，在有的场合可以使用字符串来代替正则表达式，此外，还有其他方案可以代替正则表达式。例如，在某项目中需要分析PHP代码，分离出对应的函数调用（以及源代码对应的位置）。虽然这使用正则表达式也可以实现，但无论从效率还是代码复杂度方面考虑，这都不是最优的方式。PHP已经内置解析器的接口PHP Tokenizer.

使用PHP　Tokenizer能简单、高效、准确地分析出PHP源代码的组成。token__get__all函数参数为一段PHP代码，可提取出这段代码里的常量、变量、类名、函数等。这在编写phpdoc、代码优化提速、自动加载类时都可以用到。比如，在解析URL时没必要用正则表达式，使用prase__url函数即可；在获取HTTP头时，也可以使用get__headers函数。

在进行输入校验时，可以使用PHP　5提供的filter函数。例如，校验E-mail地址的代码如下：

```
filter__var（' admin@example.com' , FILTER__VALIDATE__EMAIL）;
```

这样是不是好多了呢？如果在JavaScript里，可以使用DOM代替一些正则匹配。

这里总结几种正则表达式的取代方案，它们能部分取代正则表达式的实现。

PHP的字符串函数；

PHP的Tokenizer系列函数；

PHP的url函数及一些http函数；

PHP的filter系列函数；

JavaScript的DOM模型。

3.7 本章小结

本章中主要讲解了正则表达式的一些基本概念和语法，并且通过理论结合实际的方式，讲解了正则表达式在开发中的应用，最后介绍了正则表达式的效率优化和一些替代方案。

正则表达式中最容易混淆的就是转义，最难理解的就是环视，而环视在复杂的正则匹配中又是无法避免的，所以重点要掌握环视的概念和应用。

在实例讲解中以XSS攻击为例，强调数据过滤的必要性，“一切输入都是不可信任的”，在设计中，要始终保持高度警惕。当然，正则表达式只是一种解决方案。接下来着重讲了SEO中的静态化，通过Apache的URL重写把动态请求地址映射到一个静态HTML文件上，从而实现对搜索引擎友好的地址。

正则优化的关键理念就是“减少回溯”，常见的手段就是减少分支、使用环视和懒惰匹配。最后列举几种正则表达式的替代方案，供读者参考。

正则表达式是一种抽象语法，要熟练掌握正则表达式，不但需要学会总结，还需要多多练习。

第4章　PHP网络技术及应用

作为专注于Web编程的PHP而言，简单的网络模型和接口，使得在PHP中实现套接字、cURL等都变得极其简单。本章主要从HTTP协议入手，逐步深入，展示PHP网络技术的实际应用。并且展开讨论Socket、网络协议分析、Cookie和Session等相关知识。

HTTP协议是整个Web的基础，是客户端和服务器端协同工作的基石，要想了解Web的工作原理、优化Web应用，就要完全理解HTTP协议。

4.1　HTTP协议详解

简单来说，HTTP就是一个基于应用层的通信规范：双方要进行通信，大家都要遵守一个规范——HTTP协议。HTTP协议从WWW服务器传输超文本到本地浏览器，可以使浏览器更加高效。HTTP协议不仅保证计算机正确快速地传输超文本文档，还能确定传输文档中的哪一部分，以及哪部分内容首先显示（如文本先于图形）等。

4.1.1　HTTP协议与SPDY协议

HTTP（Hyper Text Transfer Protocol，超文本传输协议）是万维网协会（World Wide Web Consortium）和Internet工作小组（Internet Engineering Task Force, IETF）合作的结果，最终发布了一系列的RFC（Request For Comments）。RFC 1945定义了HTTP 1.0版本，最著名的是RFC 2616，其中定义了今天普遍使用的版本——HTTP 1.1。

HTTP是一个应用层协议，由请求和响应构成，是一个标准的客户端服务器模型。HTTP通常承载于TCP协议之上，有时也承载于TLS或SSL协议层之上，这个时候，就成了常说的HTTPS。默认HTTP的端口号为80，HTTPS的端口号为443。

HTTP协议在OSI模型中的位置如图4-1所示。

HTTP协议的模型就是客户端发起请求，服务器回送响应，如图4-2所示。HTTP协议是一个无状态的协议，同一个客户端的这次请求和上

次请求没有对应关系。

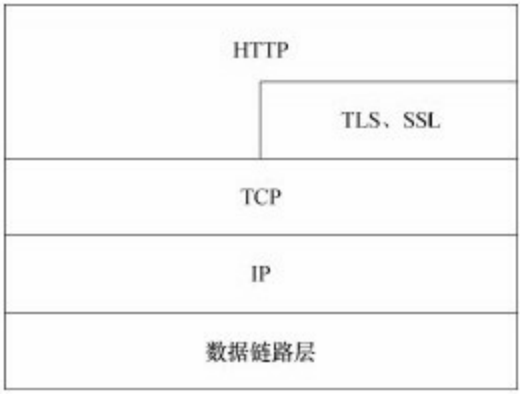

图　4-1　HTTP协议在OSI模型中的位置

图　4-2　HTTP请求响应模型

这种设计属于典型的“问答式”交互，客户端和服务器端一问一答，使HTTP协议模型异常简单。这种设计也存在问题，比如服务器端不会主动向客户端PUSH，一问一答的轮询也会使TCP连接频繁建立和断开，导致其交互效率不高。基于以上缺点，SPDY协议应运而生。

SPDY协议是Google推出的协议，优化了浏览器和服务器之间的通信，支持流复用，具备优先级的请求、主动发起请求、强制SSL安全传输等先进的特性。目前，Chrome和Firefox浏览器的最新版均支持SPDY协议，一些服务器端软件也纷纷开始支持SPDY协议，如Jetty　8、Nginx 1.3.x等服务器的最新版本。可以预见，在未来的几年，SPDY协议将得到很大推广。

SPDY协议的应用需要客户端浏览器和服务器端同时支持。目前，应用SPDY协议的主要是Google产品，如Google Plus。

4.1.2 HTTP协议如何工作

浏览网页是HTTP协议的主要应用，但是这并不代表HTTP协议就只能应用于网页的浏览。只要通信的双方都遵守HTTP协议，其就有用武之地。比如腾讯QQ、迅雷等软件都使用HTTP协议（当然还包括其他的协议）。

那么HTTP协议是如何工作的呢？

首先，客户端发送一个请求（Request）给服务器，服务器在接收到这个请求后将生成一个响应（Response）返回给客户端。一次HTTP操作称为一个事务，其工作过程可分为四步：

1）客户机与服务器需要建立连接。单击某个超链接，HTTP协议的工作开始。

2）建立连接后，客户机发送一个请求给服务器。格式为：前边是统一资源标识符（URL）、中间是协议版本号，后边是MIME信息（包括请求修饰符、客户机信息和可能的内容）。

3）服务器接到请求后，给予相应的响应信息。格式为：首先是一个状态行（包括信息的协议版本号、一个成功或错误的代码），然后是MIME信息（包括服务器信息、实体信息和可能的内容）。

4）客户端接收服务器返回的信息并显示在用户的显示屏上，然后客户机与服务器断开连接。

如果以上过程中的某一步出现错误，产生错误的信息将返回到客户端，由显示屏输出。对于用户来说，这些过程是由HTTP协议自己完成的，用户只要用鼠标单击，等待信息显示就可以了。

提示　怎样才能看到HTTP协议呢？可以查看RFC　2616文档，或者使用抓包软件。通用的抓包软件主要有IRIS、Wireshark等，专门抓取HTTP包的软件主要有HttpWatch、IE Analy zer、Fiddler、Charles等。本书使用Fiddler进行查看和调试，其体积小巧、功能强大，并且是免费的。在浏览器中使用Firefox的扩展Firebug查看HTTP请求。

下面简单介绍HTTP协议中一些主要的概念。

1.请求

在发起请求前，需要先建立连接。

连接是一个传输层的实际环流，它建立在两个相互通信的应用程序之间。在HTTP 1.1协议中，request和response头中都有可能出现一个connection的头，其决定当Client和Server通信时对于长链接如何进行处理。

HTTP 1. 1协议中，Client和Server默认对方支持长链接，如果Client使用HTTP 1.1协议，但又不希望使用长链接，需要在header中指明connection的值为close；如果Server方也不想支持长链接，则在response中需要明确说明connection的值为close。不论request还是response的header中包含了值为close的connection，都表明当前正在使用的TCP连接在请求处理完毕后会被断掉，以后Client再进行新的请求时必须创建新的TCP连接。

HTTP请求由三部分组成：请求行、消息报头、请求正文。请求行以一个方法符号开头，以空格分开，后面跟着请求的URI和协议的版本，格式如下：

```
Method Request-URI HTTP-Version CRLF
```

上述格式中各参数说明如下：

Method：请求方法。

Request URI：一个统一资源标识符。

HTTP Version：请求的HTTP协议版本。

CRLF：回车和换行（除了作为结尾的CRLF外，不允许出现单独的CR或LF字符）。

请求方法（所有方法全为大写）有多种，各个方法的解释如下：

GET：请求获取Request URI所标识的资源。

POST：在Request URI所标识的资源后附加新的数据。

HEAD：请求获取由Request URI所标识的资源的响应消息报头。

PUT：请求服务器存储一个资源，并用Request URI作为其标识。

DELETE：请求服务器删除Request URI所标识的资源。

TRACE：请求服务器回送收到的请求信息，主要用于测试或诊断。

CONNECT：保留以备将来使用。

OPTIONS：请求查询服务器的性能，或者查询与资源相关的选项和需求。

2.响应

在接收和解释请求消息后，服务器返回一个HTTP响应消息。HTTP响应也由三个部分组成，分别是：状态行、消息报头、响应正文。

状态行格式如下：

```
HTTP-Version Status-Code Reason-Phrase CRLF
```

上述格式中各参数说明如下：

HTTP Version：服务器HTTP协议的版本。

Status Code：服务器发回的响应状态代码。

Reason Phrase：状态代码的文本描述。

状态代码由三位数字组成，第一个数字定义了响应的类别，有五种可能取值：

1xx：指示信息——请求已接收，继续处理。

2xx：成功——请求已被成功接收、理解、接受。

3xx：重定向——要完成请求必须进行更进一步的操作。

4xx：客户端错误——请求有语法错误或请求无法实现。

5xx：服务器端错误——服务器未能实现合法的请求。

常见状态代码、状态描述和说明如下：

200 OK：客户端请求成功。

400 Bad Request：客户端请求有语法错误，不能被服务器所理解。

401 Unauthorize：请求未经授权，这个状态代码必须和WWW Authenticate报头域一起使用。

403 Forbidden：服务器收到请求，但是拒绝提供服务。

404 Not Found：请求资源不存在，例如输入了错误的URL。

500 Internal Server Error：服务器发生不可预期的错误。

503 Server Unavailable：服务器当前不能处理客户端的请求，一段时间后可能恢复正常。

例如，响应信息“HTTP/1.1 200 OK（CRLF）”，表示响应请求到达服务器后被成功识别，返回成功标记。响应正文就是服务器返回的资源的内容。

3.报头

HTTP消息报头包括普通报头、请求报头、响应报头、实体报头。每个报头域组成形式如下：

名字+：+空格+值

注意　消息报头域的名字是不区分英文大小写的。报头都是自解释的，具体在这里就不描述了。

1）普通报头中有少数报头域用于所有的请求和响应消息，但并不用于被传输的实体，只用于传输的消息（如缓存控制、连接控制等）。

2）请求报头允许客户端向服务器端传递请求的附加信息以及客户端自身的信息（如UA头、Accept等）。

3）响应报头允许服务器传递不能放在状态行中的附加响应信息，以及关于服务器的信息和对Request URI所标识的资源进行下一步访问的信息（如Location）。

4）实体报头定义了关于实体正文和请求所标识的资源的元信息，例如有无实体正文。

一个报头的信息截图如图4-3所示。

图 4-3 一个HTTP请求的响应数据

比较重要的几个报头如下。

Host：头域指定请求资源的Internet主机和端口号，必须表示请求URL的原始服务器或网关的位置。HTTP 1.1请求必须包含主机头域，否则系统会以400状态码返回。

User Agent：简称UA，内容包含发出请求的用户信息。通常UA包含浏览者的信息，主要是浏览器的名称版本和所用的操作系统。在图4-3所示中可以看到，客户端使用的是Gecko渲染引擎的浏览器，这里是Firefox；操作系统为Windows NT 6.1的内核，即Win dows 7操作系统

（内核版本号和操作系统代号不是一一对应的）。这个UA头不仅仅是使用浏览器才存在，只要使用了基于HTTP协议的客户端软件都会发送这个请求，无论是手机端还是PDA等。这个UA头是辨别客户端所用设备的重要依据。

Accept：告诉服务器可以接受的文件格式。通常这个值在各种浏览器中都差不多。不过WAP浏览器所能接受的格式要少一些，这也是用来区分WAP和计算机浏览器的主要依据之一。随着WAP浏览器的升级，其已经和计算机浏览器越来越接近，因此这个判断所起的作用也越来越弱。

Cookie：Cookie分两种，一种是客户端向服务器端发送的，使用Cookie报头，用来标记一些信息；另一种是服务器发送给浏览器的，报头为Set Cookie。二者的主要区别是Cookie报头的value里可以有多个Cookie值，并且不需要显式指定domain等。而Set Cookie报头里一条记录只能有一个Cookie的value，需要指明domain、path等。

Cache Control：指定请求和响应遵循的缓存机制。在请求消息或响应消息中设置Cache Control并不会修改另一个消息处理过程中的缓存处理过程。请求时的缓存指令包括no cache、no store、max age、max stale、min fresh、only if cached；响应消息中的指令包括public、private、no cache、no store、no transform、must revalidate、proxy revalidate、max age。

Referer：头域允许客户端指定请求URI的源资源地址，这可以允许服务器生成回退链表，可用来登录、优化缓存等。也允许废除的或错误的连接由于维护的目的被追踪。如果请求的URI没有自己的URI地址，Referer不能被发送。如果指定的是部分URI地址，则此地址应该是一个相对地址。Referer通常是流量统计系统用来记录来访者地址的参数。

Content Length：内容长度。

Content Range：响应的资源范围。可以在每次请求中标记请求的资源范围，在连接断开重连时，客户端只请求该资源未下载的部分，而不是重新请求整个资源，实现断点续传。迅雷就是基于这个原理，使用多线程分段读取网络上的资源，最后再合并。

Accept Encoding：指定所能接受的编码方式。通常服务器会对页面

进行GZIP压缩后再输出以减少流量，一般浏览器均支持对这种压缩后的数据进行处理。但对于我们来说，如果不想接收到这些看似“乱码”的数据，可以指定不接受任何服务器端压缩处理，要求其原样返回。

自定义报头：在HTTP消息中，也可以使用一些在HTTP 1.1正式规范里没有定义的头字段，这些头字段统称为自定义的HTTP头或者扩展头。比如server字段，在图4-3中，Google使用的是GWS服务器。这个报头一般是由服务器发送的。也可以定义一些“不正规”的报头，如“WEBMASTER：chen@qq.com”。在PHP里，使用header函数即可实现。图4-3中，X XSS Protection也是Google自定义的报头。

4.1.3　HTTP应用：模拟灌水机器人

垃圾评论和机器人一直是各大论坛和博主最头疼的问题，这些来势凶猛的数据是怎么提交到我们的服务器上的？是手工提交还是另有秘密武器？如果是用软件实现的，那么其实现原理是什么，又应该怎么防止？

为了解决这些头疼的问题，我们有必要先了解其产生的过程，然后有针对性地进行防御。知己知彼，百战不殆。

1.浏览器工作流程

其实，浏览器就是一个实现了HTTP协议的客户端软件，浏览器的工作流程如图4-4所示。

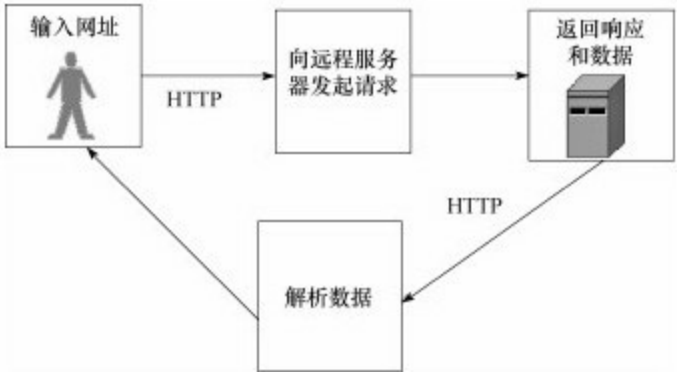

图　4-4　浏览器的工作流程

在整个过程中，浏览器扮演的角色始终是一个忠实的执行者。据此，我们很容易就能想到，只要遵循HTTP协议和服务器进行交互，就实现了一个最简单的浏览器，这个浏览器只提供对数据的收发而不提供解析功能。而对于服务器端的代码而言，是无法判断是真正的浏览器还是只是一个虚拟的浏览器。

2.PHP中和HTTP相关的函数

那怎么发送HTTP请求呢？可以使用代码发送HTTP头，如服务器端的PHP、客户端的AJAX，也可以使用各种抓包软件构造HTTP　Request包。

在这之前，先介绍一些PHP中与HTTP协议相关的一系列函数。

array get__headers（stringurl[，intformat]）函数：取得服务器响应一个HTTP请求所发送的所有标头。通常用此函数请求一个URL，根据其返回的数据判断状态码是否为200，即可判断所请求的资源是否存在。

file系列函数：包括fopen、file__get__contents等，可以用来操作文件，也可以请求一个网络上的资源。

stream__*系列函数：发送请求，包括但不限于HTTP协议。

socket系列函数：通过Socket发送和请求数据，包括但不限于HTTP协议。

cURL扩展库：PHP的一个扩展，这是一个封装的函数库。可以用来模拟浏览器和服务器进行交互，功能比较强大。

header函数：PHP中可用此函数发送原始的HTTP头。需要注意的是，这个函数之前不能有输出以及空格等。

这里用最简单的方式，即使用PHP中的http__build__query和file系列函数发送HTTP请求。

我们经常用file__get__contents打开文件，实际上用这个函数还可打开一个网络地址，实现简单的网页抓取。用file__get__contents或者fopen、file、readfile等函数读取URL的时候，会创建一个$http__response__header变量保存HTTP响应的报头，使用fopen等函数打开的数据流信息可以用stream__get__meta__data获取。简单的HTTP协议如代码清单4-1所示。

代码清单4-1 简单的HTTP协议使用示例

```
<? php
$html=file__get__contents（' http：//www.baidu.com/' ）；
print__r（$http__response__header）；
$fp=fopen（' http：//www.baidu.com/' , ' r' ）；
print__r（stream__get__meta__data（$fp））；
fclose（$fp）；
? >
```

3.模拟灌水机器人

PHP　5中新增的context参数使这些函数更加灵活，通过该参数可以定制HTTP请求，甚至POST数据。我们就利用这个函数向服务器发送请求。下面模拟一个机器人向一个博客发送留言。

首先打开某博客页面，查看评论表单的字段，为发帖机器人做准备，如图4-5所示。

从图4-5可以看出，一共有三个字段要填写，且表单使用POST方法提交。构建表单值并提交，如代码清单4-2所示。

图　4-5　表单字段

代码清单4-2　构造HTTP灌水机器人

```
<?php
$data=array('author'=>'白菜大侠','mail'=>'info@aiyooyoo.com','text'=>'博主很给力。');
$data=http_build_query($data);
$opts=array(
'http'=>array(
'method'=>'POST',
'header'=>"Content-type:application/x-www-form-urlencoded\r\n".
"Content-Length:".strlen($data)."\r\n",
'content'=>$data)
);
$context=stream_context_create($opts);
$html=
@file_get_contents('http://aiyooyoo.com/index.php/archives/7/comment',false,$context);
?>
```

这段代码遵循HTTP请求的格式构造了一段报文。注明了请求方式为POST，请求内容为data。

注意　http_build_query函数并不是必需的，这个函数仅仅是把传入的数组元素用&符号连接起来并编码，也可以自己手工构造。这里使用这个函数仅仅是为了方便而已。

运行上面的代码，刷新留言板。我们发现留言并没有提交，怎么回事？代码错了吗？还是构造的请求不符合HTTP协议规范？这里使用

Firefox下最优秀的调试工具Firebug进行测试。打开Firebug，在真实环境下发个评论，看真实环境中页面向服务器提交了什么，如图4-6所示。

图　4-6　Firebug抓到的数据

从图4-6中可以看到，真实页面向服务器提交的数据如下：

```
author=%E7%99%BD%E8%8F%9C&mail=info%40aiyooyoo.com&url=&
text=%E6%88%91%E5%B0%B1%E8%A6%81%E6%9D%A5%E7%81%8C%E6%B0%B4%EF%BC%8C%E8%B0%81%E4%B9%9F%E6%8C%A1%E4%B8%8D%E4%BD%8F%7E%7E
```

这一串就是当前页面向服务器传送的真实数据（注意：中文和特殊字符被编码了）。从HTML代码我们知道，URL是一个非必需字段，可写可不写，所以可以确认POST参数这部分是没有问题的。

但是这还不够，因为POST发送HTTP数据可能还需要Cookie、UA头等，服务器才能识别，判断结果是“这才是我要的数据”。至于是手工提交的还是机器提交的，它是分不清的。看来上述问题可能是刚才构造的数据不够完整造成的。真实的header部分如图4-7所示。

图　4-7　Firebug中的请求头信息

把有用的数据添加到header部分以实现灌水机器人，如代码清单4-3所示。

代码清单4-3　灌水机器人的实现

```
<? php
$data=array（' author' =>' 白菜大侠' , ' mail' =>' info@aiyooyoo.com' , ' text' =>' 博主很给力。' ）；
$data=http__build__query（$data）；
$opts=array（
' http' =>array（
' method' =>' POST' ,
' header' =>"Content-type：application/x-www-form-urlencoded\r\n".
"Content-Length：".strlen（$data）."\r\n".
"Cookie：PHPSESSID=15vg6jsbbj5n0cjlg7pbioai85"."\r\n".
"User-Agent：Mozilla/5.0（Windows；U；Windows NT 6.1；zh-CN；rv：1.9.2.13）Gecko/
20101203 Firefox/3.6.13"."\r\n".
"Referer：http：//aiyooyoo.com/index.php/archives/7/"."\r\n",
' content' =>$data）
）；
$context=stream__context__create（$opts）；
$html=@file__get__contents（' http：//aiyooyoo.com/index.php/archives/7/comment' , false，$context）；
? >
```

Cookie中加了当前页面所必需的参数，还加了UA、Referer等参数。

提示　至于为什么要加这些参数、要加多少、为什么不用加Accept参数等问题都是经过使用总结出来的。因为很多参数实际上都是固定的而且非必需的，只提交最主要的参数即可。

以上代码不一定是完美的，仅作演示而已。现在再来看看效果，果然很“给力”，确实可以发送了。一个粗糙的灌水机器人就制作成功了，如图4-8所示。

整个过程耗时不到5分钟。可见file__get__contents是个好东西，如果需要做简单的页面抓取和数据提交，可以考虑用这个函数。复杂的应用则需要使用cURL。但它们本质是一样的，都需要了解HTTP协议。

4.使用抓包软件构造和提交HTTP请求

以Fiddler为例看看使用抓包软件怎么构造和提交HTTP请求。

假设已经安装了Fiddler，这款软件安装时需要.NET运行时环境，Vista以上系统自带.NET运行时环境。在真实的环境下发表一个评论，如图4-9所示。

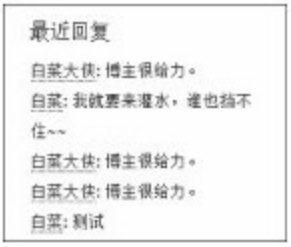

图　4-8　灌水机器人运行效果

图　4-9　表单示意

打开Fiddler，默认是捕获所有数据包，选菜单File→Capture　Traffic命令，取消默认的捕获状态。

提示　这一步操作不是必需的，主要是为了停止捕获不需要的数据包，减少后续的工作量。

由于发表帖子是发送请求，并且是操作的起点，所以这个请求通常就是第一个数据包。找到此数据包，在右边查看Raw（原始数据），由此可确认，这就是刚才发出的HTTP请求，如图4-10所示。

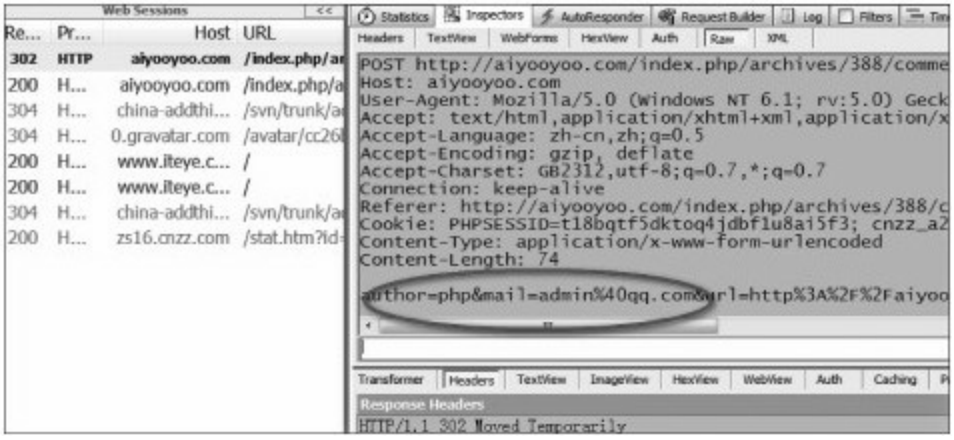

图　4-10　Fiddler抓取到的数据

在右边选择“Request　Builder”的TAB选项，打开构造请求界面，选择Raw方式，即提交原始的HTTP请求，把上一步的数据复制到数据框

中，如图4-11所示。

图　4-11　Fiddler中查看原始HTTP数据

单击“Execute”，数据包即发送到远程客户端。为了加强效果，可以多提交几次。打开网站后台看一下是否真的发送成功了，效果如图4-12所示。

图　4-12　程序运行结果

可以看到，我们发送的数据已经完全被服务器端接受了。

提示　使用代码提交HTTP请求比较麻烦，但是对流程的控制更强。比如用for循环不停地发送请求，效果更好。后面一种方式操作简

单，不需要编码，对原始数据的操控性更强。二者各有优势，综合使用能逐渐从应用中了解HTTP协议。

4.1.4 垃圾信息防御措施

由上面的讲解就可以知道垃圾评论的大体来源了，那么怎么防御这样的攻击呢？

总结一下，防止这类垃圾评论与机器人的攻击的手段如下：

1）IP限制。其原理在于IP难以伪造。即使是对于拨号用户，虽然IP可变，但这也会大大增加攻击的工作量。

2）验证码。其重点是让验证码难于识别，对于“字母+数字”的验证码，关键在于形变与重叠，增加其破解中切割和字模比对的难度，人眼尚且难以辨识，机器就更难处理了，再者是加大对于验证码的猜测难度。

3）Token和表单欺骗。通过加入隐藏的表单值或者故意对程序混淆表单值，进而达到判断是真实的用户还是软件提交的目的。

4）审核机制。加大了管理人员的工作量，但理论上可以完全阻止垃圾评论，这是最无奈也是最有效的策略。

1.IP限制

HTTP协议是透明的、公开的，服务器端根本无法区分来源是真实的提交还是伪造的。所以通过判断Referer等手段是于事无补的，但是HTTP也有自己的局限。由于HTTP协议是应用层的协议，是基于TCP/IP协议的，故一些底层的东西HTTP协议是无法伪造的，比如IP。

IP在TCP层传递，其传输需要通过TCP的“三次握手”。在这个握手过程中，有一个校验过程，因此IP是很难伪造的。而HTTP层属于顶层，无法控制IP，所以最简单有效的办法就是对IP进行限制。

但是，不少代码会判断HTTP头中的HTTP__X__FORWARDED__FOR是否是代理过来的，若是，则把其记为IP。但是，HTTP__X__FORWARDED__FOR来自于HTTP请求中的X Forwarded For报头，而这个报头是可以修改的。这就可能造成潜在的攻击。所以合理的判断是完全不考虑代理，而使用SERVER变量中的HTTP__CLIENT__IP或

REMOTE__ADDR。二者是难以伪造的。出于安全考虑，凡是通过代理过来的请求或者HTTP头中包含X Forwarded For报头的，很多程序采取了拒绝的方案。这种处理方式比较简单，误差也较小。

2.Token法

要防止攻击关键就在于加大攻击的难度。最简单的是放一个隐藏可变的Token，每次提交都需要和服务器校对，若通不过，则为外部提交。

下面的代码称为Token法，可以阻止一些简单的重复，如代码清单4-4所示。

代码清单4-4 token.php

```
<? php
define（' SECRET' , "67%$#ap28"）;
function m__token（） {
$str=mt__rand（1000, 9999）;
$str2=dechex（$__SERVER[' REQUEST__TIME' ]-$str）;
return$str.substr（md5（$str.SECRET）, 0, 10）.$str2;
}
echo m__token（）;
echo' <br/>' ;
function v__token（$str, $delay=300） {
//$delay表示时间延迟，在不同的程序根据业务来自行修改
$rs=substr（$str, 0, 4）;
$middle=substr（$str, 0, 14）;
$rs2=substr（$str, 14, 8）;
return（$middle==$rs.substr（md5（$rs.SECRET）, 0, 10））&&（$__SERVER[' REQUEST__TIME' ]-hexdec
（$rs2）-$rs<$delay）;
}
var__dump（v__token（m__token（）））;
? >
```

这样，就造成了其构造HTTP请求的困难，使其难于通用。

3.验证码

对于预防一些灌水机器人，主要采取验证码一类的措施，包括中文验证码、回答验证，但是对于这两点，专业的灌水机器人都可以突破。但是对于技术含量不高的，可以起到一定的阻止作用，同时也降低了用户体验。

灌水机器人通常都会抓取页面的FROM表单，分析必填项与非必填项，对于必填项构造请求。如果存在普通的图形验证码，则启用图形识别模块、破解验证码、构造完整的数据包。对此，可以建立一个INPUT，用CSS或者间接地通过JavaScript设其页面为不可见，如果服务器收到的数据中包含有这个控件的值，那么肯定是来自机器提交。这样

也能起到一定的效果。另外，对INPUT的值进行欺骗对博客的垃圾评论能起到一定的作用。例如：

用户名：或者可以做成图片防止机器的学习。

＜input type=text name=email/＞实际上代表用户名。

＜input type=text name=url/＞实际上是E-mail。

＜input type=text name=author/＞实际上是URL，而且是隐藏的不能有任何输入的字段。

在服务器端，如果__POST[′email′]匹配的是一个电子邮件地址，那么其一定是灌水机器人。如果author字段有值，那么其也一定是灌水机器人。这叫做机器学习欺骗。同理，可以定期变更action的提交地址，加大灌水机器人的学习难度，也可以把所有数据改为后台审核。

但是，经过灌水机器人的学习和模型修改，过一段时间必将卷土重来。阻止外部提交一直是个难题。对于表单，因为它是可见的，所以很难阻止机器的猜解和模拟。目前，可行的办法是使用Active控件，使用对称加密和服务器通信，因为Active是不透明的。但是这样也存在问题，主要是IE Only和技术难度高，大多数的网站无法使用这样的技术，特别是个人站点。

使用JavaScript进行加密验证和非平衡图形验证码，可以阻止99%的外部提交。腾讯的大部分产品都使用了这种技术，比如微博和邮箱。图4-13所示为移动139邮箱注册页面验证码。

图　4-13　移动139邮箱注册页面的验证码

移动的验证码比较有创意，其答案是四选一，猜中的概率为25%，

这似乎并不能有效地阻止机器人的猜测。但是由于其需要手机验证码的参与，这是很难伪造的，或者说伪造成本很大，因此有很高的安全性。

Matlab中文论坛的注册验证问题如图4-14所示。

图　4-14　Matlab中文论坛的注册验证码

验证码看似用户不友好，但由于其本身就是专业论坛，“不专业”的注册会员并非其所需要的，因此不存在用户不友好这一说法。

提示　这两种验证机制都是可取的。有心的读者或许会联想到网络上流传的“最变态的验证码”系列图片，思路其实是一样的。

4.2 抓包工具

在介绍HTTP协议的时候涉及了一些抓包工具，并且简单地使用它进行了HTTP协议的学习。本节将进一步学习抓包工具的使用，并利用本节所学的知识介绍模拟登录和采集的开发思路和过程。

4.2.1 抓包工具分类

按照软件的功能，我们把抓包工具分为两类：

常规抓包工具：以IRIS、Wireshark为代表，这类软件可以抓取到整个局域网内所有的数据包，主要工作在数据传输层。

专用抓包工具：只抓取某一类协议，通常工作在应用层，最常见的就是对HTTP协议的抓取，如HttpWatch、Fiddler、IE Analyzer、Charles等。

由于Fiddler功能丰富，支持HTTP断点调试，并且是免费软件（作者一直在维护和更新），是所有同类软件中更新最及时、体积最小、功能最强大的免费软件之一。本节重点介绍Fiddler。

4.2.2 Fiddler功能与原理

Fiddler是用C#编写的免费HTTP/HTTPS网络调试器，以代理服务器的方式监听系统的网络数据流动。运行Fiddler后，就会在本地打开8888端口，网络数据流通过Fiddler进行中转时，可以监视HTTP/HTTPS数据流的记录并加以分析，甚至还可以修改发送和接收的数据。

Fiddler提供了清除IE缓存、请求构造器、文本编码转换工具等一系列工具，对前端开发工作很有价值。其工作原理是在浏览器（或者其他使用HTTP协议的进程）和服务器之间扮演代理的角色，这样所有的通信都要经过它。Fiddler工作原理如图4-15所示。

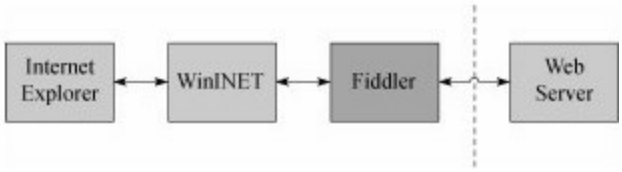

图 4-15 Fiddler工作原理

Fiddler以8888端口开本地代理服务器，并且支持HTTPS。所以，只要你的HTTP通信将代理设置为本地8888，Fiddler都能帮助你截获数据，然后中转给服务器或客户端。其最大的一个特点是可以中途修改HTTP通信内容。

提示 Sniffer和Fiddler的工作原理是一样的，但工作的网络层不同。

4.2.3 安装Fiddler

Fiddler下载地址：http：//www.fiddler2.com/fiddler2/version.asp，约660KB，当前最新版本是V2.3.9.0。Fiddler支持Windows 2000/XP/2003/Vista/Windows 7操作系统，但需要电脑安装Microsoft.NET Framework v2.0以上版本的.NET运行库。高版本的操作系统如Windows 7已经自带了.NET运行库，如果是Windows XP、Windows 2003等，需要自行安装或更新本地.NET运行库。

下载Fiddler后双击安装程序，按照提示安装到指定无路即可。安装完成后运行软件，在浏览器中输入“http：//127.0.0.1：8888/”，如果输出如图4-16所示界面，说明安装成功。

图 4-16 Fiddler运行截图

如果此时浏览网页，Fiddler就会抓取到你和服务器之间交流的数据包，如图4-17所示。

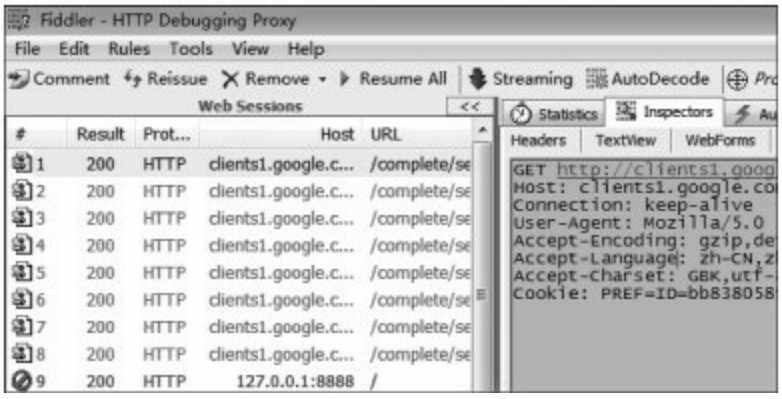

图 4-17 Fiddler抓取到的数据

打开Fiddler后，抓包默认是启用的。通常不需要特殊设置，在各种浏览器下Fiddler都能顺利工作。实际上，Fiddler和浏览器无关。读者在自己本机使用Fiddler时，可能抓取不到某个浏览器下的包，这不是Fiddler本身的问题，而是浏览器设置的问题。这种情况下，只需检查浏览器的代理设置。例如设置Firefox代理，选择“使用系统代理设置”，其他浏览器设置同理，如图4-18所示。

图　4-18　设置Firefox代理

4.2.4 Fiddler基本界面

Fiddler最基本的功能就是抓包与观察数据，下面简单介绍其界面和使用方法。

Fiddler的界面分为左右两栏：

左边为Web Sessions记录（注意，这里的Session表示一次HTTP对话，和PHP里提到的一般意义上的Session不是一个概念，这里只是沿用软件中的称呼），记录每个数据包的序列号、Host、URL、资源类型、HTTP状态码、缓存状态等基本信息。

右边划分为上下两部分，上部分为请求数据，下部分为响应数据。通常在左端的Session列表里找到请求的URL，单击即可在右边看到请求数据的详细信息。常用的三种查看方式如下：

Header（header头格式）。

Raw（HTTP协议标准格式）。

Hex View（十六进制数据流）。

在Session栏里选择某一条请求，右击，通过弹出的快捷菜单可以对其执行保存、标记、删除、重放、注释等操作，如图4-19所示。

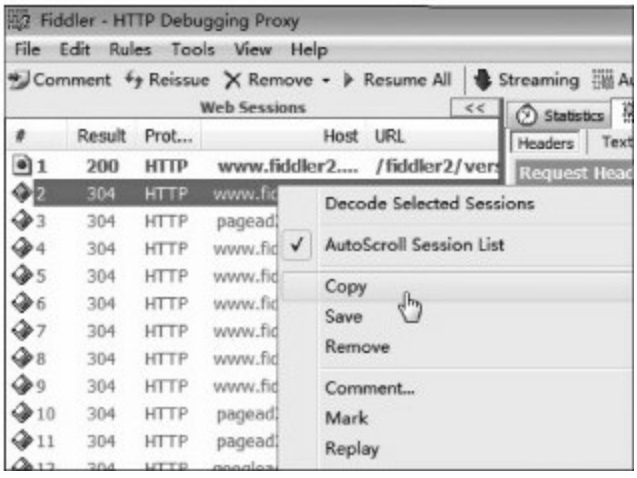

图 4-19 标记Session

下面简单介绍这几个常用功能。

保存：保存Session数据，方便以后查看或者做资源重定向。

标记：标记醒目的颜色方便查看，例如图中第一条Session标记为红色。

删除：删除不重要的或者认为可能会影响到你的Session。

重放：再次执行，这个Session的数据包将会重新发送一次。

注释：给Session添加注释。注释将显示在此Session的Comment列。

另外，在进行HTTP协议分析时，对于image、CSS类型的静态请求通常无助于我们进行协议分析，但这类Session往往还比较多，会干扰我们寻找需要的Session，故需要删除这类Session。是一条一条选择删除吗？可以这样做，但是效率不高。Fiddler提供了一个命令窗口，位于Session列表的最底部。Fiddler中内置了一些常用命令，方便管理Session，如图4-20所示。

图　4-20　Fiddler的命令窗口栏

要完成上述任务可以按以下步骤执行：

1）在命令窗口输入select image，自动选中所有image类型的Session，按delete键删除。

2）输入select css，选中所有css请求，按Delete键删除。

通过以上简单的两步，成功删除我们不需要的Session。如果当前网页有来自其他域名的Session，也会干扰我们的分析，可以输入“@google.hk.com”选中来自Google的Session，并删除。当要删除全部Session时，只需要输入“cls”并回车即可。Fiddler还支持其他命令，可以

在官方网站上找到详细的帮助文档。

我们注意到Session列表的序号前都有一个小图标，其代表什么意义呢？

表4-1列举了各类Session前面请求图标所代表的含义。

表 4-1　图标所代表的含义

图　标	含　义
	请求正在被发送到服务器
	正在从服务器下载响应
	请求被设置断点
	响应被设置断点
	请求使用了 HTTP HEAD 方法，故响应应该为空
	该请求使用了 HTTPS 加密传输
	响应体为 HTML 文档
	响应体为图像
	响应体为脚本
	响应体为 CSS 样式表
	响应体为 XML 文档
	一个普通的请求响应成功
	响应状态码是 HTTP/300，301，302，303 or 307 重定向
	响应状态码是 HTTP/304，表示使用了缓存
	响应是一个来自于客户端凭据的请求
	服务器端响应错误，如 404
	Session 异常终止

掌握了Fiddler的Session管理以及HTTP协议，即能使用好这款工具。

4.2.5 使用Fiddler进行HTTP断点调试

前面已经讲过了Fiddler的工作原理，相比其他HTTP抓包工具，Fiddler的优势就在于其特有的工作模式使其支持HTTP断点调试，这在很多场合是很有用的。下面通过一个例子学习。以Fiddler官方提供的测试页为例，其地址为：http：//www.fiddler2.com/sandbox/shop/。在这个页面，下拉框选择“1”，单击check out提交请求。这是一个再正常不过的请求了，我们的输入通过表单被传递给服务器，服务器进行处理后返回给我们。图4-21是其返回界面。

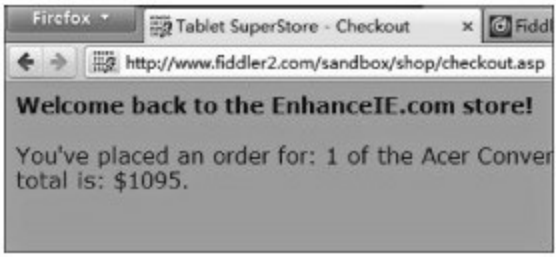

图 4-21 返回界面

再看Fiddler中所记录的请求数据包。单击左边的Session，右边窗体的数据如图4-22所示。

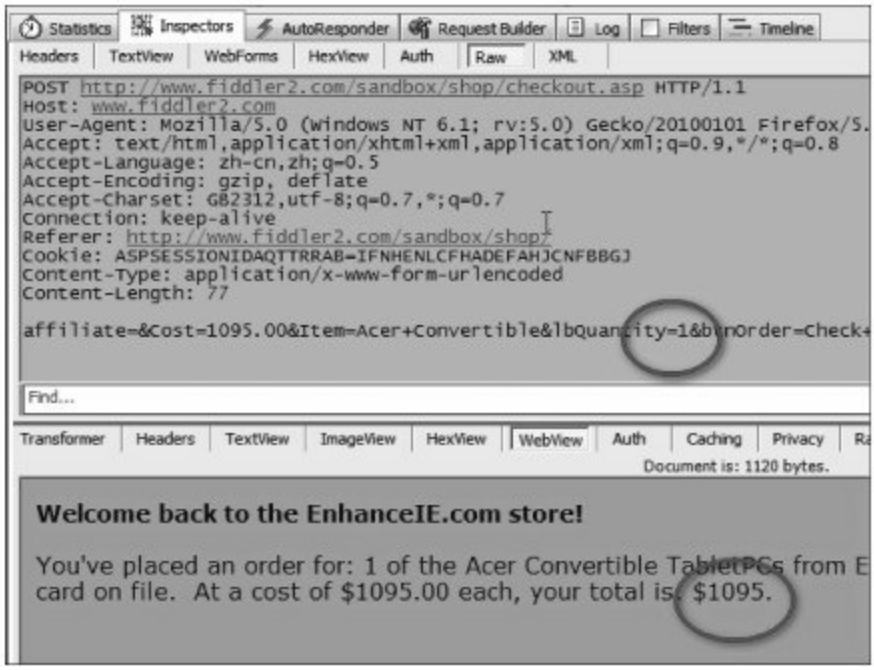

图 4-22 Fiddler抓取到的数据

注意图中标注的信息，lbQuantity是表单的参数，其值“1”是我们选择的，经服务器计算后得到结果为“1095”，并返回给浏览器。

接下来要给此响应设置断点，步骤如下：

1）在菜单中选择Rules→Automatic Breakpoints→After Responses。

2）回到订购页面，再次选择“1”，单击check out按钮。

3）现在请求发出。经过Fiddler代理发送到服务器，服务器返回响应数据到Fiddler代理。

4）此时由于设置了Responses断点，响应被挂起，就能在Fiddler中修改响应数据。

5）提交。

此时响应由代理发送到客户端，就能看到响应数据了，如图4-23所示。

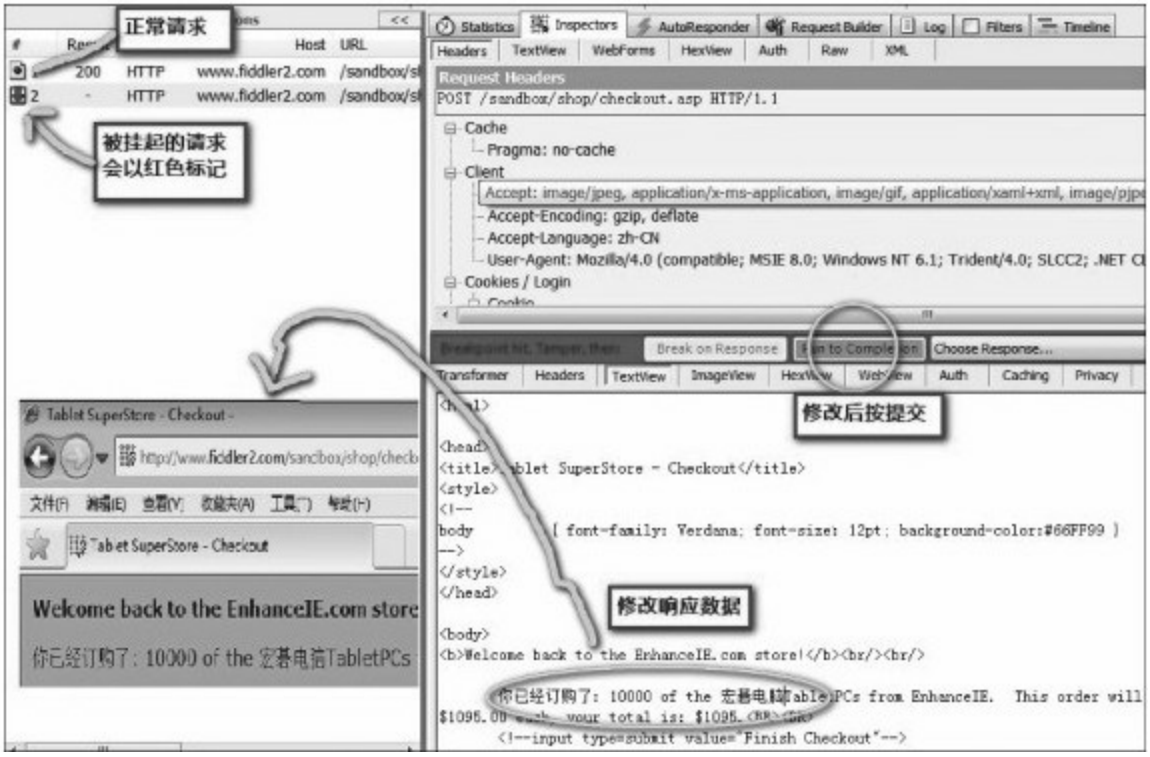

图　4-23　修改数据

可以看到，通过设置响应后（严格地说，应该是服务器响应到达Fiddler后，返回给浏览器前）断点，把HTTP发回来的响应中断并修改后才将它返回给浏览器。

还可以在请求发送前（严格地说，应该是客户端请求发送给Fiddler后，到达服务器前）设置断点，修改请求头，Fiddler代理将会把修改后的请求发送到服务器，然后读取服务器响应，中转并返回响应数据，如图4-24所示。

这两个特性对于调试AJAX程序特别有用，可以用来中断httpxml请求并修改请求内容、修改GET或者POST的数据以及header头等。这一点，也许你会觉得必要性不是太大，也许你会说：我可以自己在页面修改请求数据多次提交测试啊。但是再想想，Fiddler能够做到下面几条，这些在页面实现则较麻烦。

图　4-24　运行结果

1）修改HTTP请求的原始数据，如UA、Cookie等。

2）构造特殊请求数据。某些网页会通过使用JavaScript来限制用户在页面的数据输入。而Fiddler可以通过直接修改HTTP请求中的data突破限制，随意提交数据。

3）Fiddler通过拦截响应数据进行中断，可以修改响应体。

AJAX中有个回调函数，通常这个回调函数会根据服务器的返回值进行相应的客户端处理。如下面的代码所示：

```
$（function（）｛
$（"#student\\.idcard"）.click（function（）｛
if（$（"#student\\.card"）.val（）！=
""&&$（"#student\\.card"）.val（）.substr（0，1）==0）｛
$.get（"/order/ajax/randcard.do"，function（data）｛
switch（data.__rc）｛
case"success"：
$（"#student\\.card"）.val（data.randCard）；
break；
default：
alert（"处理失败"）；//other deal……
break；
｝
｝）；
｝
｝）；
｝）；
```

这是一段生成随机不重复流水号的代码，在输入框里输入0后单击，即可通过AJAX请求服务器，返回一段JSON数据，然后根据JSON里的data.__rc这个key的value进行不同的处理。如果是success，把返回的结果赋给该输入框，否则提示处理失败。

现在可以给该响应设置断点，修改返回的JSON数据，模拟各种情况，进而观察代码的运行情况。

上面这三个优势是不是很令你心动？你是不是灵光一闪地想到，从某种意义上来说，Fid dler这个断点功能可以帮助我们测试程序的安全性和健壮性呢？如果想到了这一点，再深入地想下去，我想你会得到更多。

除此之外，Fiddler还可以修改UA头。目前Fiddler支持对十几种浏览器UA的修改，包括IE 6～9、Firefox、Chrome、iPad、Windows Mobile以及Google的爬虫等，也可以自定义UA。使用这个功能，就能轻易地测试一些针对特定浏览器的hack，以及浏览一些需要手机才能浏览的页面。

4.3 Socket进程通信机制及应用

Socket通常称为“套接字”，用于描述IP地址和端口，是一个通信链的句柄。应用程序通过套接字向网络发出请求或者应答网络请求。Socket既不是一个程序，也不是一种协议，其只是操作系统提供的通信层的一组抽象API。

4.3.1 进程通信相关概念

进程通信的概念最初来源于单机系统。由于每个进程都在自己的地址范围内运行，为保证两个相互通信的进程之间既互不干扰又协调一致工作，操作系统为进程通信提供了相应设施，如UNIX BSD中的管道（pipe）、命名管道（named pipe）和软中断信号（signal），以及UNIX System V的消息（message）、共享存储区（shared memory）和信号量（semaphore）等，但这些都仅限于用在本机进程之间的通信。网间进程通信要解决的是不同主机进程间的相互通信问题（可把同机进程通信看作是其中的特例）。为此，首先要解决的是网间进程标识问题。同一主机上，不同进程可用唯一进程号（Process ID）标识。

网络环境下，各主机独立分配的进程号不能唯一标识该进程。例如，主机A赋予某进程号5，在B机中也可以存在5号进程，因此，“5号进程”就没有意义了。

操作系统支持的网络协议众多，不同协议的工作方式不同，地址格式也不同。因此，网间进程通信还要解决多重协议的识别问题。

为了解决上述问题，TCP/IP协议引入了下列概念。

1.端口

网络中可以被命名和寻址的通信端口，是操作系统可分配的一种资源。

按照OSI七层协议的描述，传输层与网络层在功能上的最大区别是传输层提供进程通信能力。从这个意义上讲，网络通信的最终地址就不仅仅是主机地址了，还包括可以描述进程的某种标识符。为此，TCP/IP协议提出协议端口（Protocol Port，简称端口）的概念，用于标识通信

的进程。

端口是一种抽象的软件结构（包括一些数据结构和I/O缓冲区）。应用程序（即进程）通过系统调用与某端口建立连接（binding）后，传输层传给该端口的数据都被相应进程所接收，相应进程发给传输层的数据都通过该端口输出。在TCP/IP协议的实现中，操作端口类似于一般的I/O操作，进程获取一个端口，相当于获取本地唯一I/O文件，可以用一般的读写原语访问。

类似于文件描述符，每个端口都拥有一个端口号，都是整数型标识符，用于区别不同端口。由于TCP/IP传输层的TCP协议和UDP协议是完全独立的两个软件模块，因此各自的端口号也相互独立，如TCP有一个255号端口，UDP也有一个255号端口，二者并不冲突。TCP与UDP段结构中端口的地址都是16比特，有0～65535个端口号。

对于这65536个端口号有以下使用规定：

端口号小于256的定义为常用端口，服务器一般都是通过常用端口号识别。任何TCP/IP实现所提供的服务都用1～1023之间的端口号，这是由IANA管理的。

客户端只需保证该端口号在本机上是唯一的。客户端口号因存在时间很短暂，又称临时端口号。

大多数TCP/IP实现给临时端口号分配1024～5000之间的端口号。大于5000的端口号是为其他服务器预留的。

常见的端口有FTP的21号端口，HTTP服务的80号端口，SMTP（简单邮件传输服务，后面会详细介绍）的25号端口等。

2.地址

网络通信中通信的两个进程分别处在不同的机器上。遵循以下原则：

某台主机可与多个网络相连，必须指定一个特定网络地址。

网络上每台主机应有其唯一的地址。

每台主机上的每个进程应有在该主机上的唯一标识符。

通常主机地址由网络ID和主机ID组成，在TCP/IP协议中用32位整数值表示；TCP和UDP均使用16位端口号标识用户进程。

3.连接

两个进程间的通信链路称为连接。连接表现为一些缓冲区和一组协议机制。

4.3.2 Socket演示：实现服务器端与客户端的交互

在讲解Socket的创建之前，先演示一下Socket为何物。这里用Java实现一个Socket的服务器端与客户端（也可以用任何其他支持Socket操作的语言实现），然后用PHP做为客户端请求该套接字。

在服务器端使用Socket开一个服务，端口是8001，这样就可以与多个客户端进行连接了。在客户端，向该Socket发送一条消息，服务器端在收到消息后，会根据情况进行一定的处理，返回给客户端，同时在服务器端打印所有收到的消息，如图4-25所示。

图 4-25 用Java实现一个Socket的服务器端与客户端

从图4-25中看到，服务器端和客户端都采用Java实现；一旦开启服务器端，这个服务就会被注册到Windows的网络服务中，端口为8001。使用netstat命令打印本机各端口的网络连接情况，在打印列表里看到此服务已经被注册了。一旦有客户端连接此Socket，操作系统就会为客户端自动分配一个随机端口，用来和服务器端8001端口进行通信。

既然此服务已经被注册到操作系统中，实际上此服务和腾讯QQ、FTP等是一个级别的，用它能够完成的事情很多。为了验证，使用Telnet连接，如图4-26所示。

图　4-26　Socket客户端与服务器端间的交互

由于Socket是开放的、透明的，一旦运行，任何操作Socket的语言都可以访问这个开放的服务。图4-26所示是使用Java访问Socket的，也可以使用PHP、C、Python等任何提供SocketAPI的语言访问此服务。

提示　Socket是一种服务，与其实现语言无关。基于这个性质，我们能实现不同服务之间、不同语言之间的互联互通。

代码清单4-5所示是PHP访问此Socket服务的代码。

代码清单4-5　PHP访问Socket

```
<? php
$sock=fsockopen("192.168.0.2", 8001, $errno, $errstr, 1);
if(! $sock)
{
echo"$errstr($errno)<br/>\n";
}else{
socket_set_blocking($sock, false);
fwrite($sock, "send data……\r\n"); //注意：数据末尾需要加上"\r\n"提交此请求数据，否则可能将
//无法获取服务器端的回应，即使刷新缓冲也无效，这样就只有
//等到此连接关闭时才能获取到回应
fwrite($sock, "end\r\n");
//使用end命令终止此客户端连接
while(! feof($sock)){
echo fread($sock, 128);
flush();
ob_flush();
sleep(1); }
fclose($sock);
}
```

运行结果如图4-27所示。

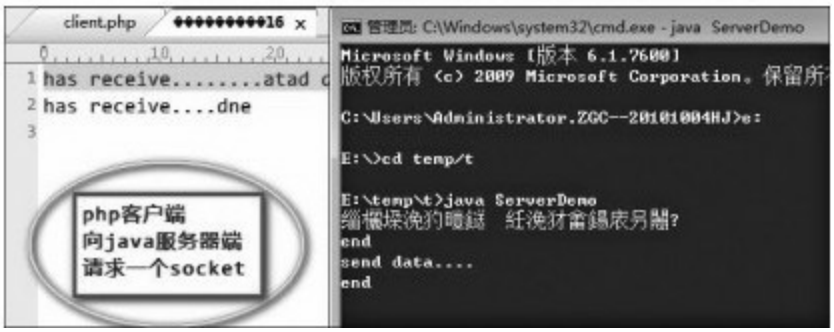

图　4-27　程序运行结果

提示　本地进程间通过TCP通信，使用Wireshark等抓包工具是抓不到数据的。上面的例子中，客户端和服务器端的数据无法抓取到。是因为回环接口的机制，这些包不会到达网卡，数据包直接被返回到传输层的输入队列中去了。而抓包工具要从网卡中获取数据。如果确实需要抓取这些数据，可以添加一条本地路由或者使用特殊抓包工具。如果是Linux系统，可以使用tcpdump命令来获取数据包。

4.3.3 Socket函数原型

看了这么多演示，大家对Socket应该有一个比较直观的认识了吧。Socket就是一种通信机制，类似于银行、电信这些部门的电话客服部门。打电话时，对方会分配置一个坐席代表回答你的问题，客服部门就相当于Socket的服务器端，你就相当于客户端。在通话结束前，如果有人想找到和你通话的坐席代表是不可能的，因为你们正在通信，客服部门的电话交换机也不会重复分配。

Socket函数的原型定义如下：

```
SOCKET socket（int af, int type, int protocol）；
```

该函数共有三个参数：

af：指定应用程序使用的通信协议的协议族，对于TCP/IP协议族该参数置AF__INET，对于UNIX可建立本地Socket。

type：指定创建的Socket类型。有三种可选项。

流套接字类型（SOCK__STREAM）：最常见的类型，基于TCP协议。

数据报套接字类型（SOCK__DGRAM）：即UDP数据报。

原始套接字类型（SOCK__RAW）：在IP层对套接字进行编程，实际上就是在IP层构造自己的IP包，然后把这个IP包发送出去。

protocol：指定应用程序所使用的通信协议。最常用的是TCP协议与UDP协议。

同样，可以把从TCP/UDP传输层过来的包抓取过来并进行分析。流套接字和数据报套接字不能完成的任务，可以在原始套接字中得以实现。所有语言提供的Socket API都是按照这个原型设计的。

提示　Socket从传输模式上又分为端对端和点对点的连接，流套接字和数据报套接字都属于端对端的连接，因此需要绑定端口号。而原始

套接字是基于IP协议的，属于点对点的传输模式，是没有端口这个概念的。比如常用的监测网络连接ping命令，就是基于ICMP协议的，它不存在端口概念。

4.3.4 PHP中的Socket函数

要创建基于Socket的应用程序，就需要详细地了解Socket的应用方法。这里以PHP为例介绍几个重要的Socket函数。

（1）resource socket__create

此函数用于创建一个Socket，代码如下：

```
resource socket__create（int $domain, int $type, int $protocol）
```

该函数共有三个参数，第一个参数指定Socket创建时所使用的通信协议族，其可选值和描述如表4-2所示。

表 4-2 通信协议族参数及描述

范 围	描 述
AF_INET	基于 IPv4 的 Internet 协议
AF_INET6	基于 IPv6 的 Internet 协议，PHP 5.0 开始添加对其的支持
AF_UNIX	UNIX 本地通信协议

第二个参数指定Socket通信的交互类型，其可选的值如表4-3所示。

表 4-3 Socket 类型参数

类 型	描 述
SOCK_STREAM	可靠的全双工链接，支持 TCP
SOCK_DGRAM	自动寻址信息功能，支持 UDP
SOCK_SEQPACKET	定序分组套接字
SOCK_RAW	构建传输层和网络层的原始套接字
SOCK_RDM	提供可信赖的数据包链接

第三个参数指定Socket使用何种类型处理协议，包括ICMP、UDP、TCP，这里不再详细介绍。

（2）socket__bind

此函数用于将IP地址和端口绑定到socket__create函数所创建的句柄中。代码如下：

```
bool socket_bind(resource$socket, string$address[, int$port=0])
```

socket_bind函数有三个参数：

第一个参数是必选参数，其值是socket_create函数所创建的句柄。

第二个参数是必选参数，其值是要绑定的IP地址。

第三个参数是可选参数，其值是要绑定的端口号，当socket_create函数所创建的第一个参数是AF_INET时，需要指定这个参数。

（3）socket_listen

在绑定Socket后，服务器端使用此函数监听客户端数据。函数原型如下：

```
bool socket_listen(resource$socket[, int$backlog=0])
```

第一个参数是socket_create函数创建的Socket句柄。

第二个参数是可选参数，表示允许的最大连接数。

（4）socket_set_block

设置为非阻塞模式。函数原型如下：

```
bool socket_set_block(resource$socket)
```

非阻塞指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。对应的概念是阻塞，阻塞就是干不完不准回来，必须得到对方的回应后才能继续下一步操作。特别是当用户比较多时，设置成非阻塞是很必要的。如果是阻塞模式，若两个客户端同时连接上，服务器端在处理一个客户端请求时，另外一个客户端的请求被会被阻塞，只有等到前一个客户端的事情处理完了，后一个客户端的请求才会被响应。

（5）socket_write

使用此函数向Socket写入数据。函数原型如下：

```
int socket__write（resource＄socket, string＄buffer[, int＄length=0]）
（6）socket__read
```

用此函数从Socket中读取指定长度的数据。函数原型如下：

```
string socket__read（resource＄socket, int＄length[, int＄type=PHP__BINARY__READ]）
```

要注意第三个参数，指定要读取数据的类型，默认为PHP__BINARY__READ，安全读取二进制数据；另外一个值是PHP__NORMAL__READ，当遇到“’r”或“＼n”时停止。

（7）pfsockopen

实现长连接。Client方与Server方先建立通信连接，连接建立后不断开，然后再进行报文发送和接收。函数原型如下：

```
pfsockopen（string＄hostname[, int＄port=-1[, int&＄errno[, string&＄errstr[, float＄timeout=ini__get
（"default__socket__timeout"）]]]]）
```

（8）socket__set__option

设置Socket的控制选项，函数原型如下：

```
bool socket__set__option（resource＄socket, int＄level, int＄optname, mixed＄optval）
```

例如设置＄socket发送超时1秒，接收超时3秒：

```
socket__set__option（＄socket, SOL__SOCKET, SO__RCVTIMEO, array（"sec"=>1, "usec"=>0））；
socket__set__option（＄socket, SOL__SOCKET, SO__SNDTIMEO, array（"sec"=>3, "usec"=>0））；
（9）socket__last__error
```

函数返回操作中任何socket函数产生的最后错误，返回值是一个int型的错误代号。函数原型如下：

```
int socket__last__error（[resource＄socket]）
```

使用socket__strerror（）函数给出对错误码的字符串描述。具体定义在Windows和类UNIX系统略有差异，限于篇幅不再列举。PHP手册中也有列举，位于：

```
\php 5.X.X\ext\sockets\unix_socket_constants.h
\php 5.X.X\ext\sockets\win32_socket_constants.h
```

提示　要想深入Socket的内部实现机制是很困难的，作为一名非底层程序员，我们只要明白Socket是一套操作系统封装好的函数，会创建和调用就可以了。

代码清单4-6演示了在PHP里创建一个Socket的方法。

代码清单4-6　PHP创建Socket

```
<? php
$host="192.168.0.2";
$port=12345;
set_time_limit（0）; //最好在CLI模式下运行，保证服务不会超时
//创建Socket
$socket=socket_create（AF_INET, SOCK_STREAM, 0）or die（"Could not create socket\n"）;
///绑定socket到指定地址和端口
$result=socket_bind（$socket, $host, $port）or die（"Could not bind to socket\n"）;
//开始监听连接
$result=socket_listen（$socket, 3）or die（"Could not set up socket listener\n"）;
//接收连接请求并调用另一个子Socket处理客户端—服务器间的信息
$spawn=socket_accept（$socket）or die（"Could not accept incoming connection\n"）;
//读取客户端输入
$input=socket_read（$spawn, 1024）or die（"Could not read input\n"）;
//clean up input string
$input=trim（$input）;
//反转客户端输入数据，返回服务端
$output=strrev（$input）."\n";
socket_write（$spawn, $output, strlen（$output））or die（"Could not write output\n"）;
//关闭sockets
socket_close（$spawn）;
socket_close（$socket）;
```

PHP的语言特性和自身定位决定了它只适合做客户端，而不适合做服务器端。因为Socket主要面向底层和网络服务开发，一般服务器端都是用C、Java等语言实现，这样能更好地操纵底层，对网络服务开发中遇到的问题（如并发、阻塞等）也有完善、成熟的解决方案，而PHP显然不适合这种应用场景。

实际上，PHP操作MySQL数据库也是通过Socket进行的，这正是由于Socket屏蔽了底层的协议，使得网络服务之间的互联互通变得简单。

提示　除了传统的服务器端语言实现的Socket外，随着HTML　5的流行，浏览器客户端实现的WebSocket也逐渐兴起，对于这一点值得关注。FlashSocket也是一个不错的解决方案。

4.3.5　Socket交互应用：使用Socket抓取数据

要在客户端操作Socket，可使用fsockopen、socket__create、stream__socket__client等函数实现。如果是PHP 5，建议使用stream__socket。

看一个实例，用Socket完成4.1.3节论坛灌水机器人的功能。

首先，到目的地（http：//typecho.org/archives/54）真实提交一次，用Fiddler查看抓到的数据，如图4-28所示。

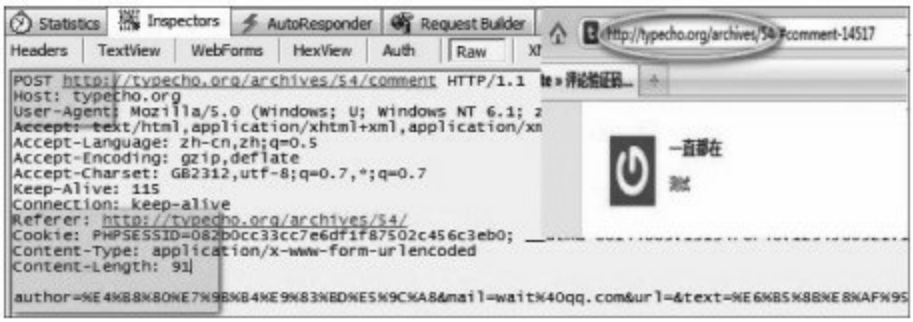

图　4-28　Fiddler抓包结果

使用Socket获取数据的实现，如代码清单4-7所示。

代码清单4-7　使用Socket获取数据

```
<? php
$post__=array（' author' =>' 一直都在' , ' mail' =>' wait@qq.com' , ' url' =>' ' , ' text' =>' 测试' ）;
$data=http__build__query（$post__）;
$fp=fsockopen（"typecho.org", 80, $errno, $errstr, 5）;
$out="POST http: //typecho.org/archives/54/comment HTTP/1.1\r\n";
$out.="Host: typecho.org\r\n";
$out.="User-Agent: Mozilla/5.0（Windows; U; Windows NT 6.1; zh-CN; rv: 1.9.2.13）Gecko/20101203 Firefox/3.6.13"."\r\n";
$out.="Content-type: application/x-www-form-urlencoded\r\n";
$out.="Referer: http: //typecho.org/archives/54/\r\n";
$out.="PHPSESSID=082b0cc33cc7e6df1f87502c456c3eb0\r\n";
$out.="Content-Length: ".strlen（$data）."\r\n";
$out.="Connection: close\r\n\r\n";
$out.=$data."\r\n\r\n";
fwrite（$fp, $out）;
while（! feof（$fp））{
echo fgets（$fp, 1280）;
}
fclose（$fp）;
```

Web应用程序是无法区分机器和人的，人和机器都是通过Socket提交数据，只不过人是通过浏览器调用操作系统的Socket提交，而机器人是通过自己写代码调用Socket提交。如果没有4.1.3节提到的防机器人设置，那么上面的代码可以很轻松地实现灌水。

最后，注意以下几点：

fsockopen的第一个参数hostname不要带“http：//”字符串，除非使用SSL等。

Headers请求不一定都要按照抓包数据全部带上，除非调试不成功或者不熟练或者有特殊需求可以全部照搬，否则只带上几个核心的header。

在Connection和data后有两个换行，如图4-28所示。

有些表单请求可能有hidden值，务必仔细抓包。

注意编码问题。

前面说过，建议使用stream__socket实现。若使用stream__socket实现只改一行代码就行，如下：

```
$fp=stream__socket__client（"tcp：//typecho.org：80"，$errno，$errstr，3）；
```

在PHP中，99.9%的Socket应用属于流套接字范畴，由于数据报套接字和原始套接字涉及比较底层的协议知识，这里就不介绍了，有兴趣的读者可以自行学习。

4.4　cURL工具及应用

在cURL还没有普及之前，常用Snoopy工具进行网络数据抓取。cURL是利用URL语法规定传输文件和数据的工具，支持很多协议，如HTTP、FTP、Telnet等。

cURL是一个通用的库，并非PHP独有。其实，很多功能用file、socket系列函数都可以实现，不过用cURL功能更全面，实现一些复杂的操作更简单，比如处理Cookie、验证、表单提交、文件上传等。

4.4.1　建立cURL请求的基本步骤

在学习更复杂的功能之前，先来看在PHP中建立cURL请求的基本步骤：

1）初始化。

2）设置选项，包括URL。

3）执行并获取HTML文档内容。

4）释放cURL句柄。

具体实现如代码清单4-8所示。

代码清单4-8　cURL的具体实现

```
<? php
//1.初始化
$ch=curl__init（）；
//2.设置选项，包括URL
curl__setopt（$ch, CURLOPT__URL，"http：//www.php.net"）；
curl__setopt（$ch, CURLOPT__RETURNTRANSFER，1）；//将curl__exec（）获取的信息以文件流的形式返回，
//而不是直接输出
curl__setopt（$ch, CURLOPT__HEADER，1）；
//启用时会将头文件的信息作为数据流输出
//3.执行并获取HTML文档内容
$output=curl__exec（$ch）；
//4.释放cURL句柄
curl__close（$ch）；
echo$output；
```

很多时候并不需要header头，把CURLOPT__HEADER设为0或者不设置（默认为0）。

第二步最重要，也就是curl__setopt（）函数，一切玄妙均在此。有一长串cURL参数可供设置，它们能指定URL请求的各个细节。

要一次性全部看完并理解可能比较困难，这里只介绍一些常用方法，详情介绍可参考PHP手册。

4.4.2　检查cURL错误和获取返回信息

我们加一段检查错误的语句（虽然这并不是必需的）：

```
//……
$output=curl_exec（$ch）；
if（$output===FALSE）{
echo"cURL Error：".curl_error（$ch）；
}
//……
```

请注意，比较时用的是“===FALSE”，而非“==FALSE”。这是为了区分空输出和布尔值FALSE，因为布尔值才是真正的错误。另外，通过curl_getinfo（）函数返回cURL执行后这一请求相关的信息，这对调试和排查错误是很有用的。代码如下所示：

```
//……
curl_exec（$ch）；
$info=curl_getinfo（$ch）；
echo′ 获取′ .$info[′ url′ ].′ 耗时′ .$info[′ total_time′ ].′ 秒′ ；
```

返回的数组如代码清单4-9所示。

代码清单4-9　返回的数组

```
（
[url]=>http：//www.php.net//资源网络地址
[content_type]=>text/html；charset=utf-8//内容编码
[http_code]=>200//http状态码
[header_size]=>395//header的大小
[request_size]=>50//请求的大小
[filetime]=>-1//文件创建时间
[ssl_verify_result]=>0//SSL验证结果
[redirect_count]=>0//跳转次数
[total_time]=>2.356//耗时
[namelookup_time]=>0//DNS查询时间
[connect_time]=>0.297//连接时间
[pretransfer_time]=>0.297//准备传输耗时
[size_upload]=>0//上传数据大小
[size_download]=>34738//下载数据大小
[speed_download]=>14744//下载速度
[speed_upload′ ]=>0//上传速度
[download_content_length]=>-1//下载内容程度
[upload_content_length]=>0//上传内容长度
[starttransfer_time]=>0.921//开始传输耗时
[redirect_time]=>0//重定向耗时
[certinfo]=>Array//认证信息
（）
）
```

这些信息在调试时是很有用的。当抓取数据出错的时候，就可以输出此数组查看具体的信息。例如，在cURL抓取的时候，可能由于网络等原因，时常出现抓取数据不完整的情况，我们就需要加一个校验。通过对所获取的数据计算filesize，然后和curl_getinfo获取的数据进行比

较，如果大小相等，就认定下载正确，否则进行重复尝试。比如，三次尝试均不成功后选择放弃或者放入失败队列，过一段时间再尝试。

看一个例子。使用cURL抓取网络上的一张图片，比较抓取图片的大小和文件头信息，目的是校验数据是否完整。详细代码如代码清单4-10所示。

代码清单4-10　cURL抓取图片

```
<? php
@header（' Content-type：image/png' ）；
//1.初始化
$ch=curl_init（）；
//2.设置选项，包括URL
curl_setopt（$ch, CURLOPT_URL，"http：//renren.com/usr/uploads/2011/06/3230341841.png"）；
curl_setopt（$ch, CURLOPT_RETURNTRANSFER，1）；//将curl_exec（）获取的信息以文件流的形式返回，
//而不是直接输出
//curl_setopt（$ch, CURLOPT_HEADER，1）；
//启用时会将头文件的信息作为数据流输出
//3.执行并获取内容
$output=curl_exec（$ch）；
//4.释放cURL句柄
$info=curl_getinfo（$ch）；
curl_close（$ch）；
file_put_contents（"g：/bak/temp/1/a.png"，$output）；
$size=filesize（"g：/bak/temp/1/a.png"）；
if（$size! =$info[' size_download' ]）{
echo' 下载数据不完整' ；
//尝试再次下载，最多三次不成功则放弃，或加入失败队列
} else {
echo' 下载数据完整，O（∩_∩）O～' ；
}
```

4.4.3　在cURL中伪造头信息

前面提到，头信息很重要，它是服务器端和客户端的身份证明和交流方式。本节用cURL模拟手机登录3g.qq.com。

首先在浏览器中输入“3g.qq.com”，会自动跳转到3gqq.qq.com，显示的不是我们要的内容。因为腾讯识别到我们在使用浏览器访问，自动转向其他地址，如图4-29所示。

图　4-29　手机腾讯网截图

要想实现手机访问的效果，就需要用cURL模拟手机UA访问它，如代码清单4-11所示。

代码清单4-11　cURL模拟手机UA

```
<? php
@header（' Content-type：text/html；charset=utf-8' ）；
//第一次初始化
$ch=curl__init（）；
curl__setopt（$ch, CURLOPT__URL，"http：//3g.qq.com"）；
curl__setopt（$ch, CURLOPT__RETURNTRANSFER，1）；
$h=array（' HTTP__VIA：HTTP/1.1 SNXA-PS-WAP-GW21（infoX-WISG, Huawei Technologies）' ，
' HTTP__ACCEPT：application/vnd.wap.wmlscriptc, text/vnd.wap.wml, application/vnd.wap.html+xml, appli
cation/xhtml+xml, text/html, multipart/mixed，*/*' ，
' HTTP__ACCEPT__CHARSET：ISO-8859-1，US-ASCII, UTF-8：Q=0.8，ISO-8859-15：Q=0.8，ISO-
10646-UCS-2：Q=0.6，UTF-16：Q=0.6' ）；
curl__setopt（$ch, CURLOPT__HTTPHEADER，$h）；
$output=curl__exec（$ch）；
curl__close（$ch）；
//第二次跳转
$ch=curl__init（）；
curl__setopt（$ch, CURLOPT__URL，"http：//info50.3g.qq.com/g/s? aid=index&amp；s__it=3&amp；g__from=
3gindex&amp；&amp；g__f=1283"）；
curl__setopt（$ch, CURLOPT__RETURNTRANSFER，1）；
curl__setopt（$ch, CURLOPT__HTTPHEADER，$h）；
$output=curl__exec（$ch）；
curl__close（$ch）；
echo$output；
```

运行结果如图4-30所示。

图　4-30　用cURL模拟手机UA访问结果

通过查看源代码可知，返回结果的确与手机设备相同。头信息来自诺基亚手机，通过访问一个输出$__SERVER的页面获得。腾讯对不同的手机型号和分辨率使用了不同的视图。比如NOKIA 1681C手机，获得的信息如下：

```
HAZZ-PS-WAP1-GW14（infoX-WISG, Huawei Technologies），HTTP__X__UP__DEVCAP__SCREENDEPTH：16 HTTP__X__UP__DEVCAP__SCREENPIXELS：128，160
```

“128，160”就是手机的屏幕尺寸。如果用宽屏手机如iPhone 4S或者安卓手机访问，由于其屏幕分辨率较大，得到的也应该是一个宽屏页面。

验证访问的页面是不是真的WAP页面，不需要看源代码，也不需要使用手机实际查看，只要更换浏览器的UA头即可，而360浏览器、Opera等都是支持的，Firefox也有对应的插件。

注意　用这种方法“欺骗”移动、联通、电信的网站是不行的。以移动为例，因为实际上我们使用真实的手机访问网站时，所有数据都是被移动网关代理的，会带上一些特殊的头，如手机号、手机特征码、手机所在基站等，这些信息很难伪造（针对我们而言是一个黑盒），普通的网站是得不到这些特殊的头的（被屏蔽）。这就是为什么现在不能通过WAP页面获取手机号的原因。

如果你的网站需要获取来访者手机号，需要和移动签订合作协议，让移动网关对你的开发服务器所在IP放开约束。

移动公司由于拥有这些数据，就能开发手机定位等Web应用。因此，我们在自己的手机上，不输入手机号就能登录移动官网查询话费。

4.4.4　在cURL中用POST方法发送数据

当发起GET请求时，数据通过“查询字串”（query　string）传递给一个URL。例如，在Google中搜索时，搜索关键字即为URL查询字串的一部分：

```
http：//www.google.com.hk/search？q=php
```

这种情况下可能并不需要cURL模拟，把这个URL丢给file__get__contents（）就能得到相同结果。

不过有一些HTML表单是用POST方法提交的。这种表单提交时，数据通过HTTP请求体（request　body）发送，而不是查询字串。当然，前面介绍过用file和Socket系列函数处理POST，这里介绍cURL处理方式。可以用PHP脚本模拟这种URL请求。

首先，新建一个可以接受并显示POST数据的文件post__output.php：

```
print__r（$__POST）；
```

接下来，写一段PHP脚本执行cURL请求，如代码清单4-12所示。

代码清单4-12　PHP中使用cURL发送数据

```
$url="http：//localhost/post__output.php"；
$post__data=array（
"foo"=>"bar",
"query"=>"php",
"action"=>"Submit"
）；
$ch=curl__init（）；
curl__setopt（$ch, CURLOPT__URL, $url）；
curl__setopt（$ch, CURLOPT__RETURNTRANSFER, 1）；
//设置为POST
curl__setopt（$ch, CURLOPT__POST, 1）；
//把POST的变量加上
curl__setopt（$ch, CURLOPT__POSTFIELDS, $post__data）；
$output=curl__exec（$ch）；
curl__close（$ch）；
echo$output；
```

执行代码后应该会得到以下结果：

```
Array
```

```
(
[foo]=>bar
[query]=>php
[action]=>Submit
)
```

这段脚本发送一个POST请求给post__output.php并返回，利用cURL捕捉了这个输出。

试着改写前面那个经典例子，向博客提交留言。现在已经使用file、Socket、cURL这三种方法实现了同一件事。

4.4.5　使用cURL上传文件

上传文件和POST十分相似，因为所有的文件上传表单都是通过POST方法提交的。首先新建一个接收文件的页面upload__output.php：

```
print_r（$_FILES）；
```

代码清单4-13所示是真正执行文件上传任务的脚本。

代码清单4-13　cURL上传文件

```
$url="http：//localhost/upload__output.php"；
$post__data=array（
"foo"=>"bar"，
//要上传的本地文件地址
"upload"=>"@test.zip"
）；
$ch=curl__init（）；
curl__setopt（$ch，CURLOPT__URL，$url）；
curl__setopt（$ch，CURLOPT__RETURNTRANSFER，1）；
curl__setopt（$ch，CURLOPT__POST，1）；
curl__setopt（$ch，CURLOPT__POSTFIELDS，$post__data）；
$output=curl__exec（$ch）；
curl__close（$ch）；
echo$output；
```

如果要上传一个文件，只需要把文件路径当作一个POST变量传过去，不过记得在前面加上@符号。执行这段脚本应该会得到如下输出：

```
Array
（
[upload]=>Array
（
[name]=>test.zip
[type]=>application/octe-stream
[tmp__name]=>aabb.tmp
[error]=>0
[size]=>118364
））
```

提示　当POST的数据来自外部时，需要注意检查并过滤@符号，因为@符号在cURL中是有特殊作用的，而本身并不属于危险字符。

4.4.6　cURL批处理

cURL还有一个高级特性——批处理句柄（handle）。这一特性允许同时或异步地打开多个cURL连接。代码清单4-14所示是来自手册的示例代码。

代码清单4-14　cURL批处理示例代码

```
//创建两个cURL资源
$ch1=curl_init（）；$ch2=curl_init（）；
//指定URL和适当的参数
curl_setopt（$ch1，CURLOPT_URL，"http：//lxr.php.net/"）；
curl_setopt（$ch1，CURLOPT_HEADER，0）；
curl_setopt（$ch2，CURLOPT_URL，"http：//www.php.net/"）；
curl_setopt（$ch2，CURLOPT_HEADER，0）；
//创建cURL批处理句柄
$mh=curl_multi_init（）；
//加上前面两个资源句柄
curl_multi_add_handle（$mh，$ch1）；
curl_multi_add_handle（$mh，$ch2）；
//预定义一个状态变量
$active=null；
//执行批处理do｛
$mrc=curl_multi_exec（$mh，$active）；
｝while（$mrc==CURLM_CALL_MULTI_PERFORM）；
while（$active&&$mrc==CURLM_OK）｛
if（curl_multi_select（$mh）！=-1）｛
do｛
$mrc=curl_multi_exec（$mh，$active）；
｝while（$mrc==CURLM_CALL_MULTI_PERFORM）；
｝｝
//关闭各个句柄
curl_multi_remove_handle（$mh，$ch1）；
curl_multi_remove_handle（$mh，$ch2）；
curl_multi_close（$mh）；
```

这里要做的就是打开多个cURL句柄并指派给一个批处理句柄，然后只需在一个while循环里等它执行完毕。

这个示例中有两个主要循环：

第一个do……while循环重复调用curl_multi_exec（）。这个函数是无隔断（non　blocking）的，但会尽可能少地执行。它返回一个状态值，只要这个值等于常量CURLM_CALL_MULTI_PER　FORM，就代表还有一些刻不容缓的工作要做（例如，把对应URL的HTTP头信息发送出去）。也就是说，需要不断调用该函数，直到返回值发生改变。

接下来的while循环，只在active变量为true时继续。这一变量之前作为第二个参数传给了curl_multi_exec（），代表只要批处理句柄中是否还有活动连接。接着调用curl_multi_select（），在活动连接（例如接受服务器响应）出现之前，它都是被“屏蔽”的。这个函数成功执行后，又会进入另一个do……while循环，继续下一条URL。

说明　很多人把这种方式称为cURL多线程处理，而curl__multi__exec并不是多线程，它属于异步处理的范畴。

4.4.7 cURL设置项

cURL有许多设置选项，这些选项才是cURL的灵魂。通过curl__setopt函数设置，原型如下：

```
bool curl__setopt（resource$ch, int$option, mixed$value）
```

由于选项特别多，我们只介绍几个最常见而又很重要的。如果需要实现SSL传输、断点传输或者遇到获取页面出现乱码、欲获取服务器超时出错等，那么下面的选项对你会有很大帮助，如表4-4所示。

表 4-4 cURL 常用选项

选　项	描　述
CURLOPT_AUTOREFERER	当根据 Location：重定向时，自动设置 header 中的 Referer：信息
CURLOPT_COOKIESESSION	启用时 cURL 会仅仅传递一个 Session Cookie，忽略其他 Cookie，默认状况下 cURL 会将所有 Cookie 返回给服务器端。Session Cookie 指用来判断服务器端的 Session 是否有效而存在的 Cookie
CURLOPT_FOLLOWLOCATION	启用将服务器返回的"Location：" 放在 header 中，递归地返回给服务器，使用 CURLOPT_MAXREDIRS 可以限定递归返回的数量
CURLOPT_HEADER	启用时将头文件的信息作为数据流输出
CURLOPT_RETURNTRANSFER	将 curl_exec()获取的信息以文件流的形式返回，而不是直接输出
CURLOPT_INFILESIZE	设定上传文件的大小，单位为字节（byte）
CURLOPT_MAXCONNECTS	允许的最大连接数量，超过会通过 CURLOPT_CLOSEPOLICY 决定应该停止哪些连接
CURLOPT_MAXREDIRS	指定 HTTP 重定向的最多数量，和 CURLOPT_FOLLOWLOCATION 一起使用
CURLOPT_COOKIE	设定 HTTP 请求中"Cookie：" 部分的内容。多个 Cookie 用分号分隔，分号后带一个空格（例如，"fruit = apple；colour = red"）
CURLOPT_COOKIEFILE	包含 Cookie 数据的文件名，Cookie 文件的格式可以是 Netscape 格式，或者只是纯 HTTP 头部信息存入文件
CURLOPT_COOKIEJAR	连接结束后保存 Cookie 信息的文件
CURLOPT_ENCODING	HTTP 请求头中"Accept-Encoding：" 的值。支持的编码有"identity"，"deflate" 和 "gzip"。如果为空字符串""，请求头会发送所有支持的编码类型
CURLOPT_POSTFIELDS	全部数据使用 HTTP 协议中的"POST" 操作来发送。要发送文件，在文件名前面加上@ 前缀并使用完整路径。这个参数通过 urlencoded 后的字符串类似 para1 = val1&para2 = val2&...或使用一个以字段名为键值，字段数据为值的数组。如果 value 是一个数组，Content-Type 头将会被设置成 multipart/form-data
CURLOPT_RANGE	以"X-Y" 的形式组成，其中 X 和 Y 都是可选项获取数据的范围，单位是字节。HTTP 传输线程也支持几个这样的重复项中间用逗号分隔如"X-Y，N-M"
CURLOPT_REFERER	HTTP 请求头中"Referer：" 的内容

（续）

选　项	描　述
CURLOPT_HTTPHEADER	用来设置 HTTP 头字段的数组。数组形式如下： array(Content-type：text/plain′, Content-length：100′)
CURLOPT_FILE	设置输出文件的位置，值是一个资源类型，默认为 STDOUT（浏览器）
CURLOPT_INFILE	在上传文件的时候需要读取的文件地址，值是一个资源类型
CURLOPT_HEADERFUNCTION	设置一个回调函数，其有两个参数：第一个是 cURL 的资源句柄，第二个是输出的 header 数据。header 数据的输出必须依赖这个函数，返回已写入的数据大小
CURLOPT_WRITEFUNCTION	拥有两个参数的回调函数：第一个是参数是会话句柄，第二是 HTTP 响应头信息的字符串。使用此回调函数，将自行处理响应头信息。响应头信息是整个字符串。设置返回值为精确的已写入字符串长度。发生错误时传输线程终止

提示　如果觉得这个函数设置起来比较麻烦，使用curl__setopt__array函数可把所有设置项作为一个数组穿进去设置。

Table of Contents